Condor® Version 7.6.0 Manual

Condor Team, University of Wisconsin–Madison

April 20, 2011

CONTENTS

LICENSING AND COPYRIGHT

Apache License
Version 2.0, January 2004
http://www.apache.org/licenses/

TERMS AND CONDITIONS FOR USE, REPRODUCTION, AND DISTRIBUTION

1. Definitions.

   "License" shall mean the terms and conditions for use, reproduction, and distribution as defined by Sections 1 through 9 of this document.

   "Licensor" shall mean the copyright owner or entity authorized by the copyright owner that is granting the License.

   "Legal Entity" shall mean the union of the acting entity and all other entities that control, are controlled by, or are under common control with that entity. For the purposes of this definition, "control" means (i) the power, direct or indirect, to cause the direction or management of such entity, whether by contract or otherwise, or (ii) ownership of fifty percent (50outstanding shares, or (iii) beneficial ownership of such entity.

   "You" (or "Your") shall mean an individual or Legal Entity exercising permissions granted by this License.

   "Source" form shall mean the preferred form for making modifications, including but not limited to software source code, documentation source, and configuration files.

   "Object" form shall mean any form resulting from mechanical transformation or translation of a Source form, including but not limited to compiled object code, generated documentation, and conversions to other media types.

   "Work" shall mean the work of authorship, whether in Source or Object form, made available under the License, as indicated by a copyright notice that is included in or attached to the work (an example is provided in the Appendix below).

   "Derivative Works" shall mean any work, whether in Source or Object form, that is based on (or derived from) the Work and for which the editorial revisions, annotations, elaborations, or other modifications represent, as a whole, an original work of authorship. For the purposes of this License, Derivative Works shall not include works that remain separable from, or merely link (or bind by name) to the interfaces of, the Work and Derivative Works thereof.

   "Contribution" shall mean any work of authorship, including the original version of the Work and any modifications or additions to that Work or Derivative Works thereof, that is intentionally submitted to Licensor for inclusion in the Work by the copyright owner or by an individual or Legal Entity authorized to submit on behalf of the copyright owner. For the purposes of this definition, "submitted" means any form of electronic, verbal, or written communication sent to the Licensor or its representatives, including but not limited to communication on electronic mailing lists, source code control systems, and issue

tracking systems that are managed by, or on behalf of, the Licensor for the purpose of discussing and improving the Work, but excluding communication that is conspicuously marked or otherwise designated in writing by the copyright owner as "Not a Contribution."

"Contributor" shall mean Licensor and any individual or Legal Entity on behalf of whom a Contribution has been received by Licensor and subsequently incorporated within the Work.

2. Grant of Copyright License. Subject to the terms and conditions of this License, each Contributor hereby grants to You a perpetual, worldwide, non-exclusive, no-charge, royalty-free, irrevocable copyright license to reproduce, prepare Derivative Works of, publicly display, publicly perform, sublicense, and distribute the Work and such Derivative Works in Source or Object form.
3. Grant of Patent License. Subject to the terms and conditions of this License, each Contributor hereby grants to You a perpetual, worldwide, non-exclusive, no-charge, royalty-free, irrevocable (except as stated in this section) patent license to make, have made, use, offer to sell, sell, import, and otherwise transfer the Work, where such license applies only to those patent claims licensable by such Contributor that are necessarily infringed by their Contribution(s) alone or by combination of their Contribution(s) with the Work to which such Contribution(s) was submitted. If You institute patent litigation against any entity (including a cross-claim or counterclaim in a lawsuit) alleging that the Work or a Contribution incorporated within the Work constitutes direct or contributory patent infringement, then any patent licenses granted to You under this License for that Work shall terminate as of the date such litigation is filed.
4. Redistribution. You may reproduce and distribute copies of the Work or Derivative Works thereof in any medium, with or without modifications, and in Source or Object form, provided that You meet the following conditions:

   (a) You must give any other recipients of the Work or Derivative Works a copy of this License; and

   (b) You must cause any modified files to carry prominent notices stating that You changed the files; and

   (c) You must retain, in the Source form of any Derivative Works that You distribute, all copyright, patent, trademark, and attribution notices from the Source form of the Work, excluding those notices that do not pertain to any part of the Derivative Works; and

   (d) If the Work includes a "NOTICE" text file as part of its distribution, then any Derivative Works that You distribute must include a readable copy of the attribution notices contained within such NOTICE file, excluding those notices that do not pertain to any part of the Derivative Works, in at least one of the following places: within a NOTICE text file distributed as part of the Derivative Works; within the Source form or documentation, if provided along with the Derivative Works; or, within a display generated by the Derivative Works, if and wherever such third-party notices normally appear. The contents of the NOTICE file are for informational purposes only and do not modify the License. You may add Your own attribution notices within Derivative Works that You distribute, alongside or as an addendum to the NOTICE text from the Work, provided that such additional attribution notices cannot be construed as modifying the License.

   You may add Your own copyright statement to Your modifications and may provide additional or different license terms and conditions for use, reproduction, or distribution of Your modifications, or for any such Derivative Works as a whole, provided Your use, reproduction, and distribution of the Work otherwise complies with the conditions stated in this License.
5. Submission of Contributions. Unless You explicitly state otherwise, any Contribution intentionally submitted for inclusion in the Work by You to the Licensor shall be under the terms and conditions of this License, without any additional terms or conditions. Notwithstanding the above, nothing herein shall supersede or modify the terms of any separate license agreement you may have executed with Licensor regarding such Contributions.
6. Trademarks. This License does not grant permission to use the trade names, trademarks, service marks, or product names of the Licensor, except as required for reasonable and customary use in describing the

origin of the Work and reproducing the content of the NOTICE file.

7. Disclaimer of Warranty. Unless required by applicable law or agreed to in writing, Licensor provides the Work (and each Contributor provides its Contributions) on an "AS IS" BASIS, WITHOUT WARRANTIES OR CONDITIONS OF ANY KIND, either express or implied, including, without limitation, any warranties or conditions of TITLE, NON-INFRINGEMENT, MERCHANTABILITY, or FITNESS FOR A PARTICULAR PURPOSE. You are solely responsible for determining the appropriateness of using or redistributing the Work and assume any risks associated with Your exercise of permissions under this License.
8. Limitation of Liability. In no event and under no legal theory, whether in tort (including negligence), contract, or otherwise, unless required by applicable law (such as deliberate and grossly negligent acts) or agreed to in writing, shall any Contributor be liable to You for damages, including any direct, indirect, special, incidental, or consequential damages of any character arising as a result of this License or out of the use or inability to use the Work (including but not limited to damages for loss of goodwill, work stoppage, computer failure or malfunction, or any and all other commercial damages or losses), even if such Contributor has been advised of the possibility of such damages.
9. Accepting Warranty or Additional Liability. While redistributing the Work or Derivative Works thereof, You may choose to offer, and charge a fee for, acceptance of support, warranty, indemnity, or other liability obligations and/or rights consistent with this License. However, in accepting such obligations, You may act only on Your own behalf and on Your sole responsibility, not on behalf of any other Contributor, and only if You agree to indemnify, defend, and hold each Contributor harmless for any liability incurred by, or claims asserted against, such Contributor by reason of your accepting any such warranty or additional liability.

END OF TERMS AND CONDITIONS

CHAPTER

ONE

Overview

1.1 High-Throughput Computing (HTC) and its Requirements

For many research and engineering projects, the quality of the research or the product is heavily dependent upon the quantity of computing cycles available. It is not uncommon to find problems that require weeks or months of computation to solve. Scientists and engineers engaged in this sort of work need a computing environment that delivers large amounts of computational power over a long period of time. Such an environment is called a High-Throughput Computing (HTC) environment. In contrast, High Performance Computing (HPC) environments deliver a tremendous amount of compute power over a short period of time. HPC environments are often measured in terms of FLoating point Operations Per Second (FLOPS). A growing community is not concerned about operations per second, but operations per month or per year. Their problems are of a much larger scale. They are more interested in how many jobs they can complete over a long period of time instead of how fast an individual job can complete.

The key to HTC is to efficiently harness the use of all available resources. Years ago, the engineering and scientific community relied on a large, centralized mainframe or a supercomputer to do computational work. A large number of individuals and groups needed to pool their financial resources to afford such a machine. Users had to wait for their turn on the mainframe, and they had a limited amount of time allocated. While this environment was inconvenient for users, the utilization of the mainframe was high; it was busy nearly all the time.

As computers became smaller, faster, and cheaper, users moved away from centralized mainframes and purchased personal desktop workstations and PCs. An individual or small group could afford a computing resource that was available whenever they wanted it. The personal computer is slower than the large centralized machine, but it provides exclusive access. Now, instead of one giant computer for a large institution, there may be hundreds or thousands of personal computers. This

is an environment of distributed ownership, where individuals throughout an organization own their own resources. The total computational power of the institution as a whole may rise dramatically as the result of such a change, but because of distributed ownership, individuals have not been able to capitalize on the institutional growth of computing power. And, while distributed ownership is more convenient for the users, the utilization of the computing power is lower. Many personal desktop machines sit idle for very long periods of time while their owners are busy doing other things (such as being away at lunch, in meetings, or at home sleeping).

1.2 Condor's Power

Condor is a software system that creates a High-Throughput Computing (HTC) environment. It effectively utilizes the computing power of workstations that communicate over a network. Condor can manage a dedicated cluster of workstations. Its power comes from the ability to effectively harness non-dedicated, preexisting resources under distributed ownership.

A user submits the job to Condor. Condor finds an available machine on the network and begins running the job on that machine. Condor has the capability to detect that a machine running a Condor job is no longer available (perhaps because the owner of the machine came back from lunch and started typing on the keyboard). It can checkpoint the job and move (migrate) the jobs to a different machine which would otherwise be idle. Condor continues the job on the new machine from precisely where it left off.

In those cases where Condor can checkpoint and migrate a job, Condor makes it easy to maximize the number of machines which can run a job. In this case, there is no requirement for machines to share file systems (for example, with NFS or AFS), so that machines across an entire enterprise can run a job, including machines in different administrative domains.

Condor can be a real time saver when a job must be run many (hundreds of) different times, perhaps with hundreds of different data sets. With one command, all of the hundreds of jobs are submitted to Condor. Depending upon the number of machines in the Condor pool, dozens or even hundreds of otherwise idle machines can be running the job at any given moment.

Condor does not require an account (login) on machines where it runs a job. Condor can do this because of its *remote system call* technology, which traps library calls for such operations as reading or writing from disk files. The calls are transmitted over the network to be performed on the machine where the job was submitted.

Condor provides powerful resource management by match-making resource owners with resource consumers. This is the cornerstone of a successful HTC environment. Other compute cluster resource management systems attach properties to the job queues themselves, resulting in user confusion over which queue to use as well as administrative hassle in constantly adding and editing queue properties to satisfy user demands. Condor implements *ClassAds*, a clean design that simplifies the user's submission of jobs.

ClassAds work in a fashion similar to the newspaper classified advertising want-ads. All machines in the Condor pool advertise their resource properties, both static and dynamic, such as

available RAM memory, CPU type, CPU speed, virtual memory size, physical location, and current load average, in a *resource offer* ad. A user specifies a *resource request* ad when submitting a job. The request defines both the required and a desired set of properties of the resource to run the job. Condor acts as a broker by matching and ranking resource offer ads with resource request ads, making certain that all requirements in both ads are satisfied. During this match-making process, Condor also considers several layers of priority values: the priority the user assigned to the resource request ad, the priority of the user which submitted the ad, and desire of machines in the pool to accept certain types of ads over others.

1.3 Exceptional Features

Checkpoint and Migration. Where programs can be linked with Condor libraries, users of Condor may be assured that their jobs will eventually complete, even in the ever changing environment that Condor utilizes. As a machine running a job submitted to Condor becomes unavailable, the job can be check pointed. The job may continue after migrating to another machine. Condor's checkpoint feature periodically checkpoints a job even in lieu of migration in order to safeguard the accumulated computation time on a job from being lost in the event of a system failure, such as the machine being shutdown or a crash.

Remote System Calls. Despite running jobs on remote machines, the Condor standard universe execution mode preserves the local execution environment via remote system calls. Users do not have to worry about making data files available to remote workstations or even obtaining a login account on remote workstations before Condor executes their programs there. The program behaves under Condor as if it were running as the user that submitted the job on the workstation where it was originally submitted, no matter on which machine it really ends up executing on.

No Changes Necessary to User's Source Code. No special programming is required to use Condor. Condor is able to run non-interactive programs. The checkpoint and migration of programs by Condor is transparent and automatic, as is the use of remote system calls. If these facilities are desired, the user only re-links the program. The code is neither recompiled nor changed.

Pools of Machines can be Hooked Together. Flocking is a feature of Condor that allows jobs submitted within a first pool of Condor machines to execute on a second pool. The mechanism is flexible, following requests from the job submission, while allowing the second pool, or a subset of machines within the second pool to set policies over the conditions under which jobs are executed.

Jobs can be Ordered. The ordering of job execution required by dependencies among jobs in a set is easily handled. The set of jobs is specified using a directed acyclic graph, where each job is a node in the graph. Jobs are submitted to Condor following the dependencies given by the graph.

Condor Enables Grid Computing. As grid computing becomes a reality, Condor is already there. The technique of glidein allows jobs submitted to Condor to be executed on grid machines

in various locations worldwide. As the details of grid computing evolve, so does Condor's ability, starting with Globus-controlled resources.

Sensitive to the Desires of Machine Owners. The owner of a machine has complete priority over the use of the machine. An owner is generally happy to let others compute on the machine while it is idle, but wants it back promptly upon returning. The owner does not want to take special action to regain control. Condor handles this automatically.

ClassAds. The ClassAd mechanism in Condor provides an extremely flexible, expressive framework for matchmaking resource requests with resource offers. Users can easily request both job requirements and job desires. For example, a user can require that a job run on a machine with 64 Mbytes of RAM, but state a preference for 128 Mbytes, if available. A workstation owner can state a preference that the workstation runs jobs from a specified set of users. The owner can also require that there be no interactive workstation activity detectable at certain hours before Condor could start a job. Job requirements/preferences and resource availability constraints can be described in terms of powerful expressions, resulting in Condor's adaptation to nearly any desired policy.

1.4 Current Limitations

Limitations on Jobs which can Checkpointed Although Condor can schedule and run any type of process, Condor does have some limitations on jobs that it can transparently checkpoint and migrate:

1. Multi-process jobs are not allowed. This includes system calls such as `fork()`, `exec()`, and `system()`.
2. Interprocess communication is not allowed. This includes pipes, semaphores, and shared memory.
3. Network communication must be brief. A job *may* make network connections using system calls such as `socket()`, but a network connection left open for long periods will delay checkpointing and migration.
4. Sending or receiving the SIGUSR2 or SIGTSTP signals is not allowed. Condor reserves these signals for its own use. Sending or receiving all other signals *is* allowed.
5. Alarms, timers, and sleeping are not allowed. This includes system calls such as `alarm()`, `getitimer()`, and `sleep()`.
6. Multiple kernel-level threads are not allowed. However, multiple user-level threads *are* allowed.
7. Memory mapped files are not allowed. This includes system calls such as `mmap()` and `munmap()`.
8. File locks are allowed, but not retained between checkpoints.

9. All files must be opened read-only or write-only. A file opened for both reading and writing will cause trouble if a job must be rolled back to an old checkpoint image. For compatibility reasons, a file opened for both reading and writing will result in a warning but not an error.
10. A fair amount of disk space must be available on the submitting machine for storing a job's checkpoint images. A checkpoint image is approximately equal to the virtual memory consumed by a job while it runs. If disk space is short, a special *checkpoint server* can be designated for storing all the checkpoint images for a pool.
11. On Linux, the job must be statically linked. *condor_compile* does this by default.
12. Reading to or writing from files larger than 2 GB is not supported.

Note: these limitations *only* apply to jobs which Condor has been asked to transparently checkpoint. If job checkpointing is not desired, the limitations above do not apply.

Security Implications. Condor does a significant amount of work to prevent security hazards, but loopholes are known to exist. Condor can be instructed to run user programs only as the UNIX user nobody, a user login which traditionally has very restricted access. But even with access solely as user nobody, a sufficiently malicious individual could do such things as fill up `/tmp` (which is world writable) and/or gain read access to world readable files. Furthermore, where the security of machines in the pool is a high concern, only machines where the UNIX user root on that machine can be trusted should be admitted into the pool. Condor provides the administrator with extensive security mechanisms to enforce desired policies.

Jobs Need to be Re-linked to get Checkpointing and Remote System Calls Although typically no source code changes are required, Condor requires that the jobs be re-linked with the Condor libraries to take advantage of checkpointing and remote system calls. This often precludes commercial software binaries from taking advantage of these services because commercial packages rarely make their object code available. Condor's other services are still available for these commercial packages.

1.5 Availability

Condor is currently available as a free download from the Internet via the World Wide Web at URL http://www.cs.wisc.edu/condor/downloads-v2. Binary distributions of Condor are available for the platforms detailed in Table 1.1. A platform is an architecture/operating system combination. Condor binaries are available for most major versions of Unix, as well as Windows.

In the table, *clipped* means that Condor does not support checkpointing or remote system calls on the given platform. This means that *standard* universe jobs are not supported. Some clipped platforms will have further limitations with respect to supported universes. See section 2.4.1 on page 14 for more details on job universes within Condor and their abilities and limitations.

The Condor source code is available for public download alongside the binary distributions.

Architecture	*Operating System*
Intel x86	- RedHat Enterprise Linux 3 - RedHat Enterprise Linux 4 (Using RHEL3 binaries) - RedHat Enterprise Linux 5 - Fedora Core 1 - 11 (Using RHEL3 binaries) - Debian Linux 5.0 (lenny) - Windows 2000 Professional and Server (Win NT 5.0) (clipped) - Windows 2003 Server (Win NT 5.2) (clipped) - Windows 2008 Server (Win NT 6.0) (clipped) - Windows XP Professional (Win NT 5.1) (clipped) - Windows Vista (Win NT 6.0) (clipped) - Windows 7 (clipped) - Macintosh OS X 10.4 (clipped)
Opteron x86_64	- Red Hat Enterprise Linux 3 - Red Hat Enterprise Linux 5 - Debian Linux 5.0 (lenny)

Table 1.1: Condor Version 7.6.0 supported platforms

NOTE: Other Linux distributions likely work, but are not tested or supported.

For more platform-specific information about Condor's support for various operating systems, see Chapter 6 on page 593.

Jobs submitted to the standard universe utilize *condor_compile* to relink programs with libraries provided by Condor. Table 1.2 lists supported compilers by platform. Other compilers may work, but are not supported.

Platform	**Compiler**	**Notes**
Red Hat Enterprise Linux 3, 4, 5 on x86	gcc, g++, and g77	as shipped
Red Hat Debian Linux 5.0 (lenny) on x86 and x86_64	gcc, g++, gfortran	as shipped
Fedora Core 1 - 11 on x86	gcc, g++, and g77	as shipped

Table 1.2: Supported compilers under Condor Version 7.6.0

1.6 Contributions to Condor

The quality of the Condor project is enhanced by the contributions of external organizations. We gratefully acknowledge the following contributions.

- The Globus Alliance (http://www.globus.org), for code and assistance in developing Condor-G and the Grid Security Infrastructure (GSI) for authentication and authorization.
- The GOZAL Project from the Computer Science Department of the Technion Israel Institute of Technology (http://www.technion.ac.il/), for their enhancements for Condor's High Availability. The *condor_had* daemon allows one of multiple machines to function as the central manager for a Condor pool. Therefore, if an acting central manager fails, another can take its place.
- Micron Corporation (http://www.micron.com/) for the MSI-based installer for Condor on Windows.
- Paradyn Project (http://www.paradyn.org/) and the Universitat Autònoma de Barcelona (http://www.caos.uab.es/) for work on the Tool Daemon Protocol (TDP).

Our Web Services API acknowledges the use of gSOAP with their requested wording:

- Part of the software embedded in this product is gSOAP software. Portions created by gSOAP are Copyright (C) 2001-2004 Robert A. van Engelen, Genivia inc. All Rights Reserved.

  THE SOFTWARE IN THIS PRODUCT WAS IN PART PROVIDED BY GENIVIA INC AND ANY EXPRESS OR IMPLIED WARRANTIES, INCLUDING, BUT NOT LIMITED TO, THE IMPLIED WARRANTIES OF MERCHANTABILITY AND FITNESS FOR A PARTICULAR PURPOSE ARE DISCLAIMED. IN NO EVENT SHALL THE AUTHOR BE LIABLE FOR ANY DIRECT, INDIRECT, INCIDENTAL, SPECIAL, EXEMPLARY, OR CONSEQUENTIAL DAMAGES (INCLUDING, BUT NOT LIMITED TO, PROCUREMENT OF SUBSTITUTE GOODS OR SERVICES; LOSS OF USE, DATA, OR PROFITS; OR BUSINESS INTERRUPTION) HOWEVER CAUSED AND ON ANY THEORY OF LIABILITY, WHETHER IN CONTRACT, STRICT LIABILITY, OR TORT (INCLUDING NEGLIGENCE OR OTHERWISE) ARISING IN ANY WAY OUT OF THE USE OF THIS SOFTWARE, EVEN IF ADVISED OF THE POSSIBILITY OF SUCH DAMAGE.
- Some distributions of Condor include the Google Coredumper library (http://goog-coredumper.sourceforge.net/). The Google Coredumper library is released under these terms:

  Copyright (c) 2005, Google Inc.
  All rights reserved.

  Redistribution and use in source and binary forms, with or without modification, are permitted provided that the following conditions are met:

  - Redistributions of source code must retain the above copyright notice, this list of conditions and the following disclaimer.
  - Redistributions in binary form must reproduce the above copyright notice, this list of conditions and the following disclaimer in the documentation and/or other materials provided with the distribution.
  - Neither the name of Google Inc. nor the names of its contributors may be used to endorse or promote products derived from this software without specific prior written permission.

1.7 Contact Information

The latest software releases, publications/papers regarding Condor and other High-Throughput Computing research can be found at the official web site for Condor at http://www.cs.wisc.edu/condor.

In addition, there is an e-mail list at condor-world@cs.wisc.edu. The Condor Team uses this e-mail list to announce new releases of Condor and other major Condor-related news items. To subscribe or unsubscribe from the the list, follow the instructions at http://www.cs.wisc.edu/condor/mail-lists/. Because many of us receive too much e-mail as it is, you will be happy to know that the Condor World e-mail list group is moderated, and only major announcements of wide interest are distributed.

Our users support each other by belonging to an unmoderated mailing list targeted at solving problems with Condor. Condor team members attempt to monitor traffic to Condor Users, responding as they can. Follow the instructions at http://www.cs.wisc.edu/condor/mail-lists/.

Finally, you can reach the Condor Team directly. The Condor Team is comprised of the developers and administrators of Condor at the University of Wisconsin-Madison. Condor questions, comments, pleas for help, and requests for commercial contract consultation or support are all welcome; send Internet e-mail to condor-admin@cs.wisc.edu. Please include your name, organization, and telephone number in your message. If you are having trouble with Condor, please help us troubleshoot by including as much pertinent information as you can, including snippets of Condor log files.

1.8 Privacy Notice

The Condor software periodically sends short messages to the Condor Project developers at the University of Wisconsin, reporting totals of machines and jobs in each running Condor system. An example of such a message is given below.

The Condor Project uses these collected reports to publish summary figures and tables, such as the total of Condor systems worldwide, or the geographic distribution of Condor systems. This information helps the Condor Project to understand the scale and composition of Condor in the real world and improve the software accordingly.

The Condor Project will not use these reports to publicly identify any Condor system or user without permission. The Condor software does not collect or report any personal information about individual users.

We hope that you will contribute to the development of Condor through this reporting feature. However, you are free to disable it at any time by changing the configuration variables `CONDOR_DEVELOPERS` and `CONDOR_DEVELOPERS_COLLECTOR`, both described in section 3.3.16 of this manual.

Example of data reported:

```
This is an automated email from the Condor system
on machine "your.condor.pool.com".  Do not reply.

This Collector has the following IDs:
    CondorVersion: 6.6.0 Nov 12 2003
    CondorPlatform: INTEL-LINUX-GLIBC22

                     Machines Owner Claimed Unclaimed Matched Preempting

          INTEL/LINUX      810    52     716        37       0          5
        INTEL/WINNT50      120     5     115         0       0          0
      SUN4u/SOLARIS28      114    12      92         9       0          1
      SUN4x/SOLARIS28        5     1       0         4       0          0
                Total     1049    70     923        50       0          6

          RunningJobs                IdleJobs
                  920                    3868
```

CHAPTER

TWO

Users' Manual

2.1 Welcome to Condor

Presenting Condor Version 7.6.0! Condor is developed by the Condor Team at the University of Wisconsin-Madison (UW-Madison), and was first installed as a production system in the UW-Madison Computer Sciences department more than 15 years ago. This Condor pool has since served as a major source of computing cycles to UW faculty and students. For many, it has revolutionized the role computing plays in their research. An increase of one, and sometimes even two, orders of magnitude in the computing throughput of a research organization can have a profound impact on its size, complexity, and scope. Over the years, the Condor Team has established collaborations with scientists from around the world, and it has provided them with access to surplus cycles (one scientist has consumed 100 CPU years!). Today, our department's pool consists of more than 700 desktop Unix workstations and more than 100 Windows machines. On a typical day, our pool delivers more than 500 CPU days to UW researchers. Additional Condor pools have been established over the years across our campus and the world. Groups of researchers, engineers, and scientists have used Condor to establish compute pools ranging in size from a handful to hundreds of workstations. We hope that Condor will help revolutionize your compute environment as well.

2.2 Introduction

In a nutshell, Condor is a specialized batch system for managing compute-intensive jobs. Like most batch systems, Condor provides a queuing mechanism, scheduling policy, priority scheme, and resource classifications. Users submit their compute jobs to Condor, Condor puts the jobs in a queue, runs them, and then informs the user as to the result.

Batch systems normally operate only with dedicated machines. Often termed compute servers, these dedicated machines are typically owned by one organization and dedicated to the sole purpose of running compute jobs. Condor can schedule jobs on dedicated machines. But unlike traditional batch systems, Condor is also designed to effectively utilize non-dedicated machines to run jobs. By being told to only run compute jobs on machines which are currently not being used (no keyboard activity, low load average, etc.), Condor can effectively harness otherwise idle machines throughout a pool of machines. This is important because often times the amount of compute power represented by the aggregate total of all the non-dedicated desktop workstations sitting on people's desks throughout the organization is far greater than the compute power of a dedicated central resource.

Condor has several unique capabilities at its disposal which are geared toward effectively utilizing non-dedicated resources that are not owned or managed by a centralized resource. These include transparent process checkpoint and migration, remote system calls, and ClassAds. Read section 1.2 for a general discussion of these features before reading any further.

2.3 Matchmaking with ClassAds

Before you learn about how to submit a job, it is important to understand how Condor allocates resources. Understanding the unique framework by which Condor matches submitted jobs with machines is the key to getting the most from Condor's scheduling algorithm.

Condor simplifies job submission by acting as a matchmaker of ClassAds. Condor's ClassAds are analogous to the classified advertising section of the newspaper. Sellers advertise specifics about what they have to sell, hoping to attract a buyer. Buyers may advertise specifics about what they wish to purchase. Both buyers and sellers list constraints that need to be satisfied. For instance, a buyer has a maximum spending limit, and a seller requires a minimum purchase price. Furthermore, both want to rank requests to their own advantage. Certainly a seller would rank one offer of $50 dollars higher than a different offer of $25. In Condor, users submitting jobs can be thought of as buyers of compute resources and machine owners are sellers.

All machines in a Condor pool advertise their attributes, such as available memory, CPU type and speed, virtual memory size, current load average, along with other static and dynamic properties. This machine ClassAd also advertises under what conditions it is willing to run a Condor job and what type of job it would prefer. These policy attributes can reflect the individual terms and preferences by which all the different owners have graciously allowed their machine to be part of the Condor pool. You may advertise that your machine is only willing to run jobs at night and when there is no keyboard activity on your machine. In addition, you may advertise a preference (rank) for running jobs submitted by you or one of your co-workers.

Likewise, when submitting a job, you specify a ClassAd with your requirements and preferences. The ClassAd includes the type of machine you wish to use. For instance, perhaps you are looking for the fastest floating point performance available. You want Condor to rank available machines based upon floating point performance. Or, perhaps you care only that the machine has a minimum of 128 Mbytes of RAM. Or, perhaps you will take any machine you can get! These job attributes and requirements are bundled up into a job ClassAd.

Condor plays the role of a matchmaker by continuously reading all the job ClassAds and all the machine ClassAds, matching and ranking job ads with machine ads. Condor makes certain that all requirements in both ClassAds are satisfied.

2.3.1 Inspecting Machine ClassAds with condor_status

Once Condor is installed, you will get a feel for what a machine ClassAd does by trying the *condor_status* command. Try the *condor_status* command to get a summary of information from ClassAds about the resources available in your pool. Type *condor_status* and hit enter to see a summary similar to the following:

```
Name       Arch     OpSys        State      Activity   LoadAv Mem  ActvtyTime

adriana.cs INTEL    SOLARIS251   Claimed    Busy       1.000  64    0+01:10:00
alfred.cs. INTEL    SOLARIS251   Claimed    Busy       1.000  64    0+00:40:00
amul.cs.wi SUN4u    SOLARIS251   Owner      Idle       1.000  128   0+06:20:04
anfrom.cs. SUN4x    SOLARIS251   Claimed    Busy       1.000  32    0+05:16:22
anthrax.cs INTEL    SOLARIS251   Claimed    Busy       0.285  64    0+00:00:00
astro.cs.w INTEL    SOLARIS251   Claimed    Busy       0.949  64    0+05:30:00
aura.cs.wi SUN4u    SOLARIS251   Owner      Idle       1.043  128   0+14:40:15

                                      ...
```

The *condor_status* command has options that summarize machine ads in a variety of ways. For example,

condor_status -available shows only machines which are willing to run jobs now.

condor_status -run shows only machines which are currently running jobs.

condor_status -l lists the machine ClassAds for all machines in the pool.

Refer to the *condor_status* command reference page located on page 816 for a complete description of the *condor_status* command.

Figure 2.1 shows the complete machine ClassAd for a single workstation: alfred.cs.wisc.edu. Some of the listed attributes are used by Condor for scheduling. Other attributes are for information purposes. An important point is that *any* of the attributes in a machine ad can be utilized at job submission time as part of a request or preference on what machine to use. Additional attributes can be easily added. For example, your site administrator can add a physical location attribute to your machine ClassAds.

2.4 Road-map for Running Jobs

The road to using Condor effectively is a short one. The basics are quickly and easily learned.

```
MyType = "Machine"
TargetType = "Job"
Name = "alfred.cs.wisc.edu"
Machine = "alfred.cs.wisc.edu"
StartdIpAddr = "<128.105.83.11:32780>"
Arch = "INTEL"
OpSys = "SOLARIS251"
UidDomain = "cs.wisc.edu"
FileSystemDomain = "cs.wisc.edu"
State = "Unclaimed"
EnteredCurrentState = 892191963
Activity = "Idle"
EnteredCurrentActivity = 892191062
VirtualMemory = 185264
Disk = 35259
KFlops = 19992
Mips = 201
LoadAvg = 0.019531
CondorLoadAvg = 0.000000
KeyboardIdle = 5124
ConsoleIdle = 27592
Cpus = 1
Memory = 64
AFSCell = "cs.wisc.edu"
START = LoadAvg - CondorLoadAvg <= 0.300000 && KeyboardIdle > 15 * 60
Requirements = TRUE
Rank = Owner == "johndoe" || Owner == "friendofjohn"
CurrentRank =  - 1.000000
LastHeardFrom = 892191963
```

Figure 2.1: Sample output from *condor_status -l alfred*

Here are all the steps needed to run a job using Condor.

Code Preparation. A job run under Condor must be able to run as a background batch job. Condor runs the program unattended and in the background. A program that runs in the background will not be able to do interactive input and output. Condor can redirect console output (stdout and stderr) and keyboard input (stdin) to and from files for you. Create any needed files that contain the proper keystrokes needed for program input. Make certain the program will run correctly with the files.

The Condor Universe. Condor has several runtime environments (called a *universe*) from which to choose. Of the universes, two are likely choices when learning to submit a job to Condor: the standard universe and the vanilla universe. The standard universe allows a job running under Condor to handle system calls by returning them to the machine where the job was submitted. The standard universe also provides the mechanisms necessary to take a checkpoint and migrate a partially completed job, should the machine on which the job is executing become unavailable. To use the standard universe, it is necessary to relink the program with

the Condor library using the *condor_compile* command. The manual page for *condor_compile* on page 714 has details.

The vanilla universe provides a way to run jobs that cannot be relinked. There is no way to take a checkpoint or migrate a job executed under the vanilla universe. For access to input and output files, jobs must either use a shared file system, or use Condor's File Transfer mechanism.

Choose a universe under which to run the Condor program, and re-link the program if necessary.

Submit description file. Controlling the details of a job submission is a submit description file. The file contains information about the job such as what executable to run, the files to use for keyboard and screen data, the platform type required to run the program, and where to send e-mail when the job completes. You can also tell Condor how many times to run a program; it is simple to run the same program multiple times with multiple data sets.

Write a submit description file to go with the job, using the examples provided in section 2.5.1 for guidance.

Submit the Job. Submit the program to Condor with the *condor_submit* command.

Once submitted, Condor does the rest toward running the job. Monitor the job's progress with the *condor_q* and *condor_status* commands. You may modify the order in which Condor will run your jobs with *condor_prio*. If desired, Condor can even inform you in a log file every time your job is checkpointed and/or migrated to a different machine.

When your program completes, Condor will tell you (by e-mail, if preferred) the exit status of your program and various statistics about its performances, including time used and I/O performed. If you are using a log file for the job (which is recommended) the exit status will be recorded in the log file. You can remove a job from the queue prematurely with *condor_rm*.

2.4.1 Choosing a Condor Universe

A *universe* in Condor defines an execution environment. Condor Version 7.6.0 supports several different universes for user jobs:

- Standard
- Vanilla
- Grid
- Java
- Scheduler
- Local
- Parallel

- VM

The **universe** under which a job runs is specified in the submit description file. If a universe is not specified, the default is vanilla, unless your Condor administrator has changed the default. However, we strongly encourage you to specify the universe, since the default can be changed by your Condor administrator, and the default that ships with Condor has changed.

The standard universe provides migration and reliability, but has some restrictions on the programs that can be run. The vanilla universe provides fewer services, but has very few restrictions. The grid universe allows users to submit jobs using Condor's interface. These jobs are submitted for execution on grid resources. The java universe allows users to run jobs written for the Java Virtual Machine (JVM). The scheduler universe allows users to submit lightweight jobs to be spawned by the program known as a daemon on the submit host itself. The parallel universe is for programs that require multiple machines for one job. See section 2.9 for more about the Parallel universe. The vm universe allows users to run jobs where the job is no longer a simple executable, but a disk image, facilitating the execution of a virtual machine.

Standard Universe

In the standard universe, Condor provides *checkpointing* and *remote system calls*. These features make a job more reliable and allow it uniform access to resources from anywhere in the pool. To prepare a program as a standard universe job, it must be relinked with *condor_compile*. Most programs can be prepared as a standard universe job, but there are a few restrictions.

Condor checkpoints a job at regular intervals. A *checkpoint image* is essentially a snapshot of the current state of a job. If a job must be migrated from one machine to another, Condor makes a checkpoint image, copies the image to the new machine, and restarts the job continuing the job from where it left off. If a machine should crash or fail while it is running a job, Condor can restart the job on a new machine using the most recent checkpoint image. In this way, jobs can run for months or years even in the face of occasional computer failures.

Remote system calls make a job perceive that it is executing on its home machine, even though the job may execute on many different machines over its lifetime. When a job runs on a remote machine, a second process, called a *condor_shadow* runs on the machine where the job was submitted. When the job attempts a system call, the *condor_shadow* performs the system call instead and sends the results to the remote machine. For example, if a job attempts to open a file that is stored on the submitting machine, the *condor_shadow* will find the file, and send the data to the machine where the job is running.

To convert your program into a standard universe job, you must use *condor_compile* to relink it with the Condor libraries. Put *condor_compile* in front of your usual link command. You do not need to modify the program's source code, but you do need access to the unlinked object files. A commercial program that is packaged as a single executable file cannot be converted into a standard universe job.

For example, if you would have linked the job by executing:

```
% cc main.o tools.o -o program
```

Then, relink the job for Condor with:

```
% condor_compile cc main.o tools.o -o program
```

There are a few restrictions on standard universe jobs:

1. Multi-process jobs are not allowed. This includes system calls such as `fork()`, `exec()`, and `system()`.
2. Interprocess communication is not allowed. This includes pipes, semaphores, and shared memory.
3. Network communication must be brief. A job *may* make network connections using system calls such as `socket()`, but a network connection left open for long periods will delay checkpointing and migration.
4. Sending or receiving the SIGUSR2 or SIGTSTP signals is not allowed. Condor reserves these signals for its own use. Sending or receiving all other signals *is* allowed.
5. Alarms, timers, and sleeping are not allowed. This includes system calls such as `alarm()`, `getitimer()`, and `sleep()`.
6. Multiple kernel-level threads are not allowed. However, multiple user-level threads *are* allowed.
7. Memory mapped files are not allowed. This includes system calls such as `mmap()` and `munmap()`.
8. File locks are allowed, but not retained between checkpoints.
9. All files must be opened read-only or write-only. A file opened for both reading and writing will cause trouble if a job must be rolled back to an old checkpoint image. For compatibility reasons, a file opened for both reading and writing will result in a warning but not an error.
10. A fair amount of disk space must be available on the submitting machine for storing a job's checkpoint images. A checkpoint image is approximately equal to the virtual memory consumed by a job while it runs. If disk space is short, a special *checkpoint server* can be designated for storing all the checkpoint images for a pool.
11. On Linux, the job must be statically linked. *condor_compile* does this by default.
12. Reading to or writing from files larger than 2 GB is not supported.

Vanilla Universe

The vanilla universe in Condor is intended for programs which cannot be successfully re-linked. Shell scripts are another case where the vanilla universe is useful. Unfortunately, jobs run under the vanilla universe cannot checkpoint or use remote system calls. This has unfortunate consequences for a job that is partially completed when the remote machine running a job must be returned to its owner. Condor has only two choices. It can suspend the job, hoping to complete it at a later time, or it can give up and restart the job *from the beginning* on another machine in the pool.

Since Condor's remote system call features cannot be used with the vanilla universe, access to the job's input and output files becomes a concern. One option is for Condor to rely on a shared file system, such as NFS or AFS. Alternatively, Condor has a mechanism for transferring files on behalf of the user. In this case, Condor will transfer any files needed by a job to the execution site, run the job, and transfer the output back to the submitting machine.

Under Unix, Condor presumes a shared file system for vanilla jobs. However, if a shared file system is unavailable, a user can enable the Condor File Transfer mechanism. On Windows platforms, the default is to use the File Transfer mechanism. For details on running a job with a shared file system, see section 2.5.3 on page 23. For details on using the Condor File Transfer mechanism, see section 2.5.4 on page 25.

Grid Universe

The Grid universe in Condor is intended to provide the standard Condor interface to users who wish to start jobs intended for remote management systems. Section 5.3 on page 555 has details on using the Grid universe. The manual page for *condor_submit* on page 825 has detailed descriptions of the grid-related attributes.

Java Universe

A program submitted to the Java universe may run on any sort of machine with a JVM regardless of its location, owner, or JVM version. Condor will take care of all the details such as finding the JVM binary and setting the classpath.

Scheduler Universe

The scheduler universe allows users to submit lightweight jobs to be run immediately, alongside the *condor_schedd* daemon on the submit host itself. Scheduler universe jobs are not matched with a remote machine, and will never be preempted. The job's requirements expression is evaluated against the *condor_schedd*'s ClassAd.

Originally intended for meta-schedulers such as *condor_dagman*, the scheduler universe can also be used to manage jobs of any sort that must run on the submit host.

However, unlike the local universe, the scheduler universe does not use a *condor_starter* daemon to manage the job, and thus offers limited features and policy support. The local universe is a better choice for most jobs which must run on the submit host, as it offers a richer set of job management features, and is more consistent with other universes such as the vanilla universe. The scheduler universe may be retired in the future, in favor of the newer local universe.

Local Universe

The local universe allows a Condor job to be submitted and executed with different assumptions for the execution conditions of the job. The job does not wait to be matched with a machine. It instead executes right away, on the machine where the job is submitted. The job will never be preempted. The job's requirements expression is evaluated against the *condor_schedd*'s ClassAd.

Parallel Universe

The parallel universe allows parallel programs, such as MPI jobs, to be run within the opportunistic Condor environment. Please see section 2.9 for more details.

VM Universe

Condor facilitates the execution of VMware and Xen virtual machines with the vm universe.

Please see section 2.11 for details.

2.5 Submitting a Job

A job is submitted for execution to Condor using the *condor_submit* command. *condor_submit* takes as an argument the name of a file called a submit description file. This file contains commands and keywords to direct the queuing of jobs. In the submit description file, Condor finds everything it needs to know about the job. Items such as the name of the executable to run, the initial working directory, and command-line arguments to the program all go into the submit description file. *condor_submit* creates a job ClassAd based upon the information, and Condor works toward running the job.

The contents of a submit file can save time for Condor users. It is easy to submit multiple runs of a program to Condor. To run the same program 500 times on 500 different input data sets, arrange your data files accordingly so that each run reads its own input, and each run writes its own output. Each individual run may have its own initial working directory, stdin, stdout, stderr, command-line arguments, and shell environment. A program that directly opens its own files will read the file names to use either from stdin or from the command line. A program that opens a static filename every time will need to use a separate subdirectory for the output of each run.

The *condor_submit* manual page is on page 825 and contains a complete and full description of how to use *condor_submit*. It also includes descriptions of all the commands that may be placed into a submit description file. In addition, the index lists entries for each command under the heading of Submit Commands.

2.5.1 Sample submit description files

In addition to the examples of submit description files given in the *condor_submit* manual page, here are a few more.

Example 1

Example 1 is one of the simplest submit description files possible. It queues up one copy of the program *foo* (which had been created by *condor_compile*) for execution by Condor. Since no platform is specified, Condor will use its default, which is to run the job on a machine which has the same architecture and operating system as the machine from which it was submitted. No `input`, `output`, and `error` commands are given in the submit description file, so the files `stdin`, `stdout`, and `stderr` will all refer to `/dev/null`. The program may produce output by explicitly opening a file and writing to it. A log file, `foo.log`, will also be produced that contains events the job had during its lifetime inside of Condor. When the job finishes, its exit conditions will be noted in the log file. It is recommended that you always have a log file so you know what happened to your jobs.

```
####################
#
# Example 1
# Simple condor job description file
#
####################

Executable   = foo
Universe     = standard
Log          = foo.log
Queue
```

Example 2

Example 2 queues two copies of the program *mathematica*. The first copy will run in directory `run_1`, and the second will run in directory `run_2`. For both queued copies, `stdin` will be `test.data`, `stdout` will be `loop.out`, and `stderr` will be `loop.error`. There will be two sets of files written, as the files are each written to their own directories. This is a convenient way to organize data if you have a large group of Condor jobs to run. The example file shows

program submission of *mathematica* as a vanilla universe job. This may be necessary if the source and/or object code to *mathematica* is not available.

```
####################
#
# Example 2: demonstrate use of multiple
# directories for data organization.
#
####################

Executable = mathematica
Universe   = vanilla
input      = test.data
output     = loop.out
error      = loop.error
Log        = loop.log

Initialdir = run_1
Queue

Initialdir = run_2
Queue
```

Example 3

The submit description file for Example 3 queues 150 runs of program *foo* which has been compiled and linked for RHEL 3 running on a 32-bit Intel processor. This job requires Condor to run the program on machines which have greater than 32 megabytes of physical memory, and expresses a preference to run the program on machines with more than 64 megabytes. It also advises Condor that it will use up to 28 megabytes of memory when running. Each of the 150 runs of the program is given its own process number, starting with process number 0. So, files `stdin`, `stdout`, and `stderr` will refer to `in.0`, `out.0`, and `err.0` for the first run of the program, `in.1`, `out.1`, and `err.1` for the second run of the program, and so forth. A log file containing entries about when and where Condor runs, checkpoints, and migrates processes for all the 150 queued programs will be written into the single file `foo.log`.

```
####################
#
# Example 3: Show off some fancy features including
# use of pre-defined macros and logging.
#
####################

Executable    = foo
```

```
Universe      = standard
Requirements  = Memory >= 32 && OpSys == "LINUX" && Arch =="INTEL"
Rank          = Memory >= 64
Image_Size    = 28 Meg

Error   = err.$(Process)
Input   = in.$(Process)
Output  = out.$(Process)
Log     = foo.log

Queue 150
```

2.5.2 About Requirements and Rank

The `requirements` and `rank` commands in the submit description file are powerful and flexible. Using them effectively requires care, and this section presents those details.

Both `requirements` and `rank` need to be specified as valid Condor ClassAd expressions, however, default values are set by the *condor_submit* program if these are not defined in the submit description file. From the *condor_submit* manual page and the above examples, you see that writing ClassAd expressions is intuitive, especially if you are familiar with the programming language C. There are some pretty nifty expressions you can write with ClassAds. A complete description of ClassAds and their expressions can be found in section 4.1 on page 475.

All of the commands in the submit description file are case insensitive, *except* for the ClassAd attribute string values. ClassAd attribute names are case insensitive, but ClassAd string values are *case preserving*.

Note that the comparison operators (`<`, `>`, `<=`, `>=`, and `==`) compare strings case insensitively. The special comparison operators `=?=` and `=!=` compare strings case sensitively.

A **requirements** or **rank** command in the submit description file may utilize attributes that appear in a machine or a job ClassAd. Within the submit description file (for a job) the prefix `MY.` (on a ClassAd attribute name) causes a reference to the job ClassAd attribute, and the prefix `TARGET.` causes a reference to a potential machine or matched machine ClassAd attribute.

The *condor_status* command displays statistics about machines within the pool. The **-l** option displays the machine ClassAd attributes for all machines in the Condor pool. The job ClassAds, if there are jobs in the queue, can be seen with the *condor_q -l* command. This shows all the defined attributes for current jobs in the queue.

A list of defined ClassAd attributes for job ClassAds is given in the unnumbered Appendix on page 902. A list of defined ClassAd attributes for machine ClassAds is given in the unnumbered Appendix on page 913.

Rank Expression Examples

When considering the match between a job and a machine, rank is used to choose a match from among all machines that satisfy the job's requirements and are available to the user, after accounting for the user's priority and the machine's rank of the job. The rank expressions, simple or complex, define a numerical value that expresses preferences.

The job's `rank` expression evaluates to one of three values. It can be UNDEFINED, ERROR, or a floating point value. If `rank` evaluates to a floating point value, the best match will be the one with the largest, positive value. If no `rank` is given in the submit description file, then Condor substitutes a default value of 0.0 when considering machines to match. If the job's `rank` of a given machine evaluates to UNDEFINED or ERROR, this same value of 0.0 is used. Therefore, the machine is still considered for a match, but has no rank above any other.

A boolean expression evaluates to the numerical value of 1.0 if true, and 0.0 if false.

The following `rank` expressions provide examples to follow.

For a job that desires the machine with the most available memory:

```
Rank = memory
```

For a job that prefers to run on a friend's machine on Saturdays and Sundays:

```
Rank = ( (clockday == 0) || (clockday == 6) )
       && (machine == "friend.cs.wisc.edu")
```

For a job that prefers to run on one of three specific machines:

```
Rank = (machine == "friend1.cs.wisc.edu") ||
       (machine == "friend2.cs.wisc.edu") ||
       (machine == "friend3.cs.wisc.edu")
```

For a job that wants the machine with the best floating point performance (on Linpack benchmarks):

```
Rank = kflops
```

This particular example highlights a difficulty with rank expression evaluation as currently defined. While all machines have floating point processing ability, not all machines will have the `kflops` attribute defined. For machines where this attribute is not defined, `Rank` will evaluate to the value UNDEFINED, and Condor will use a default rank of the machine of 0.0. The `rank` attribute will only rank machines where the attribute is defined. Therefore, the machine with the highest floating point performance may not be the one given the highest rank.

So, it is wise when writing a `rank` expression to check if the expression's evaluation will lead to the expected resulting ranking of machines. This can be accomplished using the *condor_status* command with the *-constraint* argument. This allows the user to see a list of machines that fit a constraint. To see which machines in the pool have `kflops` defined, use

```
condor_status -constraint kflops
```

Alternatively, to see a list of machines where `kflops` is not defined, use

```
condor_status -constraint "kflops=?=undefined"
```

For a job that prefers specific machines in a specific order:

```
Rank = ((machine == "friend1.cs.wisc.edu")*3) +
       ((machine == "friend2.cs.wisc.edu")*2) +
        (machine == "friend3.cs.wisc.edu")
```

If the machine being ranked is `"friend1.cs.wisc.edu"`, then the expression

```
(machine == "friend1.cs.wisc.edu")
```

is true, and gives the value 1.0. The expressions

```
(machine == "friend2.cs.wisc.edu")
```

and

```
(machine == "friend3.cs.wisc.edu")
```

are false, and give the value 0.0. Therefore, `rank` evaluates to the value 3.0. In this way, machine `"friend1.cs.wisc.edu"` is ranked higher than machine `"friend2.cs.wisc.edu"`, machine `"friend2.cs.wisc.edu"` is ranked higher than machine `"friend3.cs.wisc.edu"`, and all three of these machines are ranked higher than others.

2.5.3 Submitting Jobs Using a Shared File System

If vanilla, java, or parallel universe jobs are submitted without using the File Transfer mechanism, Condor must use a shared file system to access input and output files. In this case, the job *must* be able to access the data files from any machine on which it could potentially run.

As an example, suppose a job is submitted from blackbird.cs.wisc.edu, and the job requires a particular data file called `/u/p/s/psilord/data.txt`. If the job were to run on cardinal.cs.wisc.edu, the file `/u/p/s/psilord/data.txt` must be available through either NFS or AFS for the job to run correctly.

Condor allows users to ensure their jobs have access to the right shared files by using the `FileSystemDomain` and `UidDomain` machine ClassAd attributes. These attributes specify which machines have access to the same shared file systems. All machines that mount the same shared directories in the same locations are considered to belong to the same file system domain. Similarly, all machines that share the same user information (in particular, the same UID, which is important for file systems like NFS) are considered part of the same UID domain.

The default configuration for Condor places each machine in its own UID domain and file system domain, using the full host name of the machine as the name of the domains. So, if a pool *does* have access to a shared file system, the pool administrator *must* correctly configure Condor such that all the machines mounting the same files have the same `FileSystemDomain` configuration. Similarly, all machines that share common user information must be configured to have the same `UidDomain` configuration.

When a job relies on a shared file system, Condor uses the `requirements` expression to ensure that the job runs on a machine in the correct `UidDomain` and `FileSystemDomain`. In this case, the default `requirements` expression specifies that the job must run on a machine with the same `UidDomain` and `FileSystemDomain` as the machine from which the job is submitted. This default is almost always correct. However, in a pool spanning multiple `UidDomain`s and/or `FileSystemDomain`s, the user may need to specify a different `requirements` expression to have the job run on the correct machines.

For example, imagine a pool made up of both desktop workstations and a dedicated compute cluster. Most of the pool, including the compute cluster, has access to a shared file system, but some of the desktop machines do not. In this case, the administrators would probably define the `FileSystemDomain` to be `cs.wisc.edu` for all the machines that mounted the shared files, and to the full host name for each machine that did not. An example is `jimi.cs.wisc.edu`.

In this example, a user wants to submit vanilla universe jobs from her own desktop machine (jimi.cs.wisc.edu) which does not mount the shared file system (and is therefore in its own file system domain, in its own world). But, she wants the jobs to be able to run on more than just her own machine (in particular, the compute cluster), so she puts the program and input files onto the shared file system. When she submits the jobs, she needs to tell Condor to send them to machines that have access to that shared data, so she specifies a different `requirements` expression than the default:

```
Requirements = TARGET.UidDomain == "cs.wisc.edu" && \
               TARGET.FileSystemDomain == "cs.wisc.edu"
```

WARNING: If there is *no* shared file system, or the Condor pool administrator does not configure the `FileSystemDomain` setting correctly (the default is that each machine in a pool is in its own file system and UID domain), a user submits a job that cannot use remote system calls (for example,

a vanilla universe job), and the user does not enable Condor's File Transfer mechanism, the job will *only* run on the machine from which it was submitted.

2.5.4 Submitting Jobs Without a Shared File System: Condor's File Transfer Mechanism

Condor works well without a shared file system. The Condor file transfer mechanism permits the user to select which files are transferred and under which circumstances. Condor can transfer any files needed by a job from the machine where the job was submitted into a remote scratch directory on the machine where the job is to be executed. Condor executes the job and transfers output back to the submitting machine. The user specifies which files and directories to transfer, and at what point the output files should be copied back to the submitting machine. This specification is done within the job's submit description file.

Default Behavior across Condor Universes and Platforms

The default behavior of the file transfer mechanism varies across the different Condor universes, and it differs between Unix and Windows machines.

For jobs submitted under the **standard** universe, the existence of a shared file system is not relevant. Access to files (input and output) is handled through Condor's remote system call mechanism. The executable and checkpoint files are transferred automatically, when needed. Therefore, the user does not need to change the submit description file if there is no shared file system, as the file transfer mechanism is not utilized.

For the **vanilla**, **java**, and **parallel** universes, access to input files and the executable through a shared file system is presumed as a default on jobs submitted from Unix machines. If there is no shared file system, then Condor's file transfer mechanism must be explicitly enabled. When submitting a job from a Windows machine, Condor presumes the opposite: no access to a shared file system. It instead enables the file transfer mechanism by default. Submission of a job might need to specify which files to transfer, and/or when to transfer the output files back.

For the grid universe, jobs are to be executed on remote machines, so there would never be a shared file system between machines. See section 5.3.2 for more details.

For the scheduler universe, Condor is only using the machine from which the job is submitted. Therefore, the existence of a shared file system is not relevant.

Specifying If and When to Transfer Files

To enable the file transfer mechanism, place two commands in the job's submit description file: **should_transfer_files** and **when_to_transfer_output**. In the common case, they will be set as:

```
should_transfer_files = YES
```

```
when_to_transfer_output = ON_EXIT
```

Setting the **should_transfer_files** command explicitly enables or disables the file transfer mechanism. The command takes on one of three possible values:

1. `YES`: Condor transfers both the executable and the file defined by the **input** command from the machine where the job is submitted to the remote machine where the job is to be executed. The file defined by the **output** command as well as any files created by the execution of the job are transferred back to the machine where the job was submitted. When they are transferred and the directory location of the files is determined by the command **when_to_transfer_output**.
2. `IF_NEEDED`: Condor transfers files if the job is matched with and to be executed on a machine in a different `FileSystemDomain` than the one the submit machine belongs to, the same as if `should_transfer_files = YES`. If the job is matched with a machine in the local `FileSystemDomain`, Condor will not transfer files and relies on the shared file system.
3. `NO`: Condor's file transfer mechanism is disabled.

The **when_to_transfer_output** command tells Condor when output files are to be transferred back to the submit machine. The command takes on one of two possible values:

1. `ON_EXIT`: Condor transfers the file defined by the **output** command, as well as any other files in the remote scratch directory created by the job, back to the submit machine only when the job exits on its own.
2. `ON_EXIT_OR_EVICT`: Condor behaves the same as described for the value `ON_EXIT` when the job exits on its own. However, if, and each time the job is evicted from a machine, *files are transferred back at eviction time*. The files that are transferred back at eviction time may include intermediate files that are not part of the final output of the job. Before the job starts running again, all of the files that were stored when the job was last evicted are copied to the job's new remote scratch directory.

   The purpose of saving files at eviction time is to allow the job to resume from where it left off. This is similar to using the checkpoint feature of the standard universe, but just specifying `ON_EXIT_OR_EVICT` is not enough to make a job capable of producing or utilizing checkpoints. The job must be designed to save and restore its state using the files that are saved at eviction time.

   The files that are transferred back at eviction time are not stored in the location where the job's final output will be written when the job exits. Condor manages these files automatically, so usually the only reason for a user to worry about them is to make sure that there is enough space to store them. The files are stored on the submit machine in a temporary directory within the directory defined by the configuration variable `SPOOL`. The directory is named using the `ClusterId` and `ProcId` job ClassAd attributes. The directory name takes the form:

   ```
   <X mod 10000>/<Y mod 10000>/cluster<X>.proc<Y>.subproc0
   ```

where <X> is the value of ClusterId, and <Y> is the value of ProcId. As an example, if job 735.0 is evicted, it will produce the directory

```
$(SPOOL)/735/0/cluster735.proc0.subproc0
```

There is no default value for **when_to_transfer_output**. If using the file transfer mechanism, this command must be defined. However, if **when_to_transfer_output** is specified in the submit description file, but **should_transfer_files** is not, Condor assumes a value of YES for **should_transfer_files**.

NOTE: The combination of:

```
should_transfer_files = IF_NEEDED
when_to_transfer_output = ON_EXIT_OR_EVICT
```

would produce undefined file access semantics. Therefore, this combination is prohibited by *condor_submit*.

When submitting from a Windows platform, the file transfer mechanism is enabled by default. If the two commands **when_to_transfer_output** and **should_transfer_files** are *not* in the job's submit description file, then Condor uses the values:

```
should_transfer_files = YES
when_to_transfer_output = ON_EXIT
```

Specifying What Files to Transfer

If the file transfer mechanism is enabled, Condor will transfer the following files before the job is run on a remote machine.

1. the executable, as defined with the **executable** command
2. the input, as defined with the **input** command
3. any jar files, for the **java** universe, as defined with the **jar_files** command

If the job requires other input files, the submit description file should utilize the **transfer_input_files** command. This comma-separated list specifies any other files or directories that Condor is to transfer to the remote scratch directory, to set up the execution environment for the job before it is run. These files are placed in the same directory as the job's executable. For example:

```
should_transfer_files = YES
when_to_transfer_output = ON_EXIT
transfer_input_files = file1,file2
```

This example explicitly enables the file transfer mechanism, and it transfers the executable, the file specified by the **input** command, any jar files specified by the **jar_files** command, and files `file1` and `file2`.

If the file transfer mechanism is enabled, Condor will transfer the following files from the execute machine back to the submit machine after the job exits.

1. the output file, as defined with the **output** command
2. the error file, as defined with the **error** command
3. any files created by the job in the remote scratch directory; this only occurs for jobs other than **grid** universe, and for Condor-C **grid** universe jobs; directories created by the job within the remote scratch directory are ignored for this automatic detection of files to be transferred.

A path given for **output** and **error** commands represents a path on the submit machine. If no path is specified, the directory specified with **initialdir** is used, and if that is not specified, the directory from which the job was submitted is used. At the time the job is submitted, zero-length files are created on the submit machine, at the given path for the files defined by the **output** and **error** commands. This permits job submission failure, if these files cannot be written by Condor.

To *restrict* the output files or permit entire directory contents to be transferred, specify the exact list with **transfer_output_files**. Delimit the list of file names, directory names, or paths with commas. When this list is defined, and any of the files or directories do not exist as the job exits, Condor considers this an error, and places the job on hold. When this list is defined, automatic detection of output files created by the job is disabled. Paths specified in this list refer to locations on the execute machine. The naming and placement of files and directories relies on the term *base name*. By example, the path `a/b/c` has the base name `c`. It is the file name or directory name with all directories leading up to that name stripped off. On the submit machine, the transferred files or directories are named using only the base name. Therefore, each output file or directory must have a different name, even if they originate from different paths.

For **grid** universe jobs other than than Condor-C grid jobs, files to be transferred (other than standard output and standard error) must be specified using **transfer_output_files** in the submit description file, because automatic detection of new files created by the job does not take place.

Here are examples to promote understanding of what files and directories are transferred, and how they are named after transfer. Assume that the job produces the following structure within the remote scratch directory:

```
o1
o2
d1 (directory)
    o3
    o4
```

If the submit description file sets

```
transfer_output_files = o1,o2,d1
```

then transferred back to the submit machine will be

```
o1
o2
d1 (directory)
    o3
    o4
```

Note that the directory `d1` and all its contents are specified, and therefore transferred. If the directory `d1` is not created by the job before exit, then the job is placed on hold. If the directory `d1` is created by the job before exit, but is empty, this is not an error.

If, instead, the submit description file sets

```
transfer_output_files = o1,o2,d1/o3
```

then transferred back to the submit machine will be

```
o1
o2
o3
```

Note that only the base name is used in the naming and placement of the file specified with `d1/o3`.

File Paths for File Transfer

The file transfer mechanism specifies file names and/or paths on both the file system of the submit machine and on the file system of the execute machine. Care must be taken to know which machine, submit or execute, is utilizing the file name and/or path.

Files in the **transfer_input_files** command are specified as they are accessed on the submit machine. The job, as it executes, accesses files as they are found on the execute machine.

There are three ways to specify files and paths for **transfer_input_files**:

1. Relative to the current working directory as the job is submitted, if the submit command **initialdir** is not specified.
2. Relative to the initial directory, if the submit command **initialdir** is specified.
3. Absolute.

Before executing the program, Condor copies the executable, an input file as specified by the submit command **input**, along with any input files specified by **transfer_input_files**. All these files are placed into a remote scratch directory on the execute machine, in which the program runs. Therefore, the executing program must access input files relative to its working directory. Because all files and directories listed for transfer are placed into a single, flat directory, inputs must be uniquely named to avoid collision when transferred. A collision causes the last file in the list to overwrite the earlier one.

Both relative and absolute paths may be used in **transfer_output_files**. Relative paths are relative to the job's remote scratch directory on the execute machine. When the files and directories are copied back to the submit machine, they are placed in the job's initial working directory as the base name of the original path. An alternate name or path may be specified by using **transfer_output_remaps**.

A job may create files outside the remote scratch directory but within the file system of the execute machine, in a directory such as `/tmp`, if this directory is guaranteed to exist and be accessible on all possible execute machines. However, Condor will not automatically transfer such files back after execution completes, nor will it clean up these files.

Here are several examples to illustrate the use of file transfer. The program executable is called *my_program*, and it uses three command-line arguments as it executes: two input file names and an output file name. The program executable and the submit description file for this job are located in directory `/scratch/test`.

Here is the directory tree as it exists on the submit machine, for all the examples:

```
/scratch/test (directory)
      my_program.condor (the submit description file)
      my_program (the executable)
      files (directory)
          logs2 (directory)
          in1 (file)
          in2 (file)
      logs (directory)
```

Example 1 This first example explicitly transfers input files. These input files to be transferred are specified relative to the directory where the job is submitted. An output file specified in the **arguments** command, `out1`, is created when the job is executed. It will be transferred back into the directory `/scratch/test`.

```
# file name:  my_program.condor
# Condor submit description file for my_program
Executable      = my_program
Universe        = vanilla
Error           = logs/err.$(cluster)
Output          = logs/out.$(cluster)
Log             = logs/log.$(cluster)
```

```
should_transfer_files = YES
when_to_transfer_output = ON_EXIT
transfer_input_files = files/in1,files/in2

Arguments       = in1 in2 out1
Queue
```

The log file is written on the submit machine, and is not involved with the file transfer mechanism.

Example 2 This second example is identical to Example 1, except that absolute paths to the input files are specified, instead of relative paths to the input files.

```
# file name:  my_program.condor
# Condor submit description file for my_program
Executable      = my_program
Universe        = vanilla
Error           = logs/err.$(cluster)
Output          = logs/out.$(cluster)
Log             = logs/log.$(cluster)

should_transfer_files = YES
when_to_transfer_output = ON_EXIT
transfer_input_files = /scratch/test/files/in1,/scratch/test/files/in2

Arguments       = in1 in2 out1
Queue
```

Example 3 This third example illustrates the use of the submit command **initialdir**, and its effect on the paths used for the various files. The expected location of the executable is not affected by the **initialdir** command. All other files (specified by **input**, **output**, **error**, **transfer_input_files**, as well as files modified or created by the job and automatically transferred back) are located relative to the specified **initialdir**. Therefore, the output file, `out1`, will be placed in the `files` directory. Note that the `logs2` directory exists to make this example work correctly.

```
# file name:  my_program.condor
# Condor submit description file for my_program
Executable      = my_program
Universe        = vanilla
Error           = logs2/err.$(cluster)
Output          = logs2/out.$(cluster)
Log             = logs2/log.$(cluster)

initialdir      = files

should_transfer_files = YES
when_to_transfer_output = ON_EXIT
transfer_input_files = in1,in2

Arguments       = in1 in2 out1
Queue
```

Example 4 – Illustrates an Error This example illustrates a job that will fail. The files specified using the **transfer_input_files** command work correctly (see Example 1). However, relative paths to files in the **arguments** command cause the executing program to fail. The file system on the submission side may utilize relative paths to files, however those files are placed into the single, flat, remote scratch directory on the execute machine.

```
# file name:  my_program.condor
# Condor submit description file for my_program
Executable      = my_program
Universe        = vanilla
Error           = logs/err.$(cluster)
Output          = logs/out.$(cluster)
Log             = logs/log.$(cluster)

should_transfer_files = YES
when_to_transfer_output = ON_EXIT
transfer_input_files = files/in1,files/in2

Arguments       = files/in1 files/in2 files/out1
Queue
```

This example fails with the following error:

```
err: files/out1: No such file or directory.
```

Example 5 – Illustrates an Error As with Example 4, this example illustrates a job that will fail. The executing program's use of absolute paths cannot work.

```
# file name:  my_program.condor
# Condor submit description file for my_program
Executable      = my_program
Universe        = vanilla
Error           = logs/err.$(cluster)
Output          = logs/out.$(cluster)
Log             = logs/log.$(cluster)

should_transfer_files = YES
when_to_transfer_output = ON_EXIT
transfer_input_files = /scratch/test/files/in1, /scratch/test/files/in2

Arguments = /scratch/test/files/in1 /scratch/test/files/in2 /scratch/test/files/out1
Queue
```

The job fails with the following error:

```
err: /scratch/test/files/out1: No such file or directory.
```

Example 6 This example illustrates a case where the executing program creates an output file in a directory other than within the remote scratch directory that the program executes within. The file creation may or may not cause an error, depending on the existence and permissions of the directories on the remote file system.

The output file `/tmp/out1` is transferred back to the job's initial working directory as `/scratch/test/out1`.

```
# file name:  my_program.condor
# Condor submit description file for my_program
Executable      = my_program
Universe        = vanilla
Error           = logs/err.$(cluster)
Output          = logs/out.$(cluster)
Log             = logs/log.$(cluster)

should_transfer_files = YES
when_to_transfer_output = ON_EXIT
transfer_input_files = files/in1,files/in2
transfer_output_files = /tmp/out1

Arguments       = in1 in2 /tmp/out1
Queue
```

Behavior for Error Cases

This section describes Condor's behavior for some error cases in dealing with the transfer of files.

Disk Full on Execute Machine When transferring any files from the submit machine to the remote scratch directory, if the disk is full on the execute machine, then the job is place on hold.

Error Creating Zero-Length Files on Submit Machine As a job is submitted, Condor creates zero-length files as placeholders on the submit machine for the files defined by **output** and **error**. If these files cannot be created, then job submission fails.

This job submission failure avoids having the job run to completion, only to be unable to transfer the job's output due to permission errors.

Error When Transferring Files from Execute Machine to Submit Machine When a job exits, or potentially when a job is evicted from an execute machine, one or more files may be transferred from the execute machine back to the machine on which the job was submitted.

During transfer, if any of the following three similar types of errors occur, the job is put on hold as the error occurs.

1. If the file cannot be opened on the submit machine, for example because the system is out of inodes.
2. If the file cannot be written on the submit machine, for example because the permissions do not permit it.
3. If the write of the file on the submit machine fails, for example because the system is out of disk space.

Input File Transfer Using a URL

For vanilla and vm universe jobs only, Condor has the ability to allow a job's input file to be obtained by the machine allocated to execute the job with the specification of a URL. This capability requires administrative set up, as described in section 3.13.3.

To use this feature, Condor's file transfer mechanism must be enabled. Therefore, the submit description file for the job will define both **should_transfer_files** and **when_to_transfer_output**. In addition, the URL for the any files specified this way are given in the **transfer_input_files** command. An example portion of the submit description file for a job that has a single file specified with a URL:

```
should_transfer_files = YES
when_to_transfer_output = ON_EXIT
transfer_input_files = http://www.full.url/path/to/filename
```

The destination file is given by the file name in the URL.

Requirements and Rank for File Transfer

The `requirements` expression for a job must depend on the `should_transfer_files` command. The job must specify the correct logic to ensure that the job is matched with a resource that meets the file transfer needs. If no `requirements` expression is in the submit description file, or if the expression specified does not refer to the attributes listed below, *condor_submit* adds an appropriate clause to the `requirements` expression for the job. *condor_submit* appends these clauses with a logical AND, `&&`, to ensure that the proper conditions are met. Here are the default clauses corresponding to the different values of `should_transfer_files`:

1. `should_transfer_files = YES` results in the addition of the clause `(HasFileTransfer)`. If the job is always going to transfer files, it is required to match with a machine that has the capability to transfer files.
2. `should_transfer_files = NO` results in the addition of `(TARGET.FileSystemDomain == MY.FileSystemDomain)`. In addition, Condor automatically adds the `FileSystemDomain` attribute to the job ad, with whatever string is defined for the *condor_schedd* to which the job is submitted. If the job is not using the file transfer mechanism, Condor assumes it will need a shared file system, and therefore, a machine in the same `FileSystemDomain` as the submit machine.
3. `should_transfer_files = IF_NEEDED` results in the addition of

   ```
   (HasFileTransfer || (TARGET.FileSystemDomain == MY.FileSystemDomain))
   ```

   If Condor will optionally transfer files, it must require that the machine is *either* capable of transferring files *or* in the same file system domain.

To ensure that the job is matched to a machine with enough local disk space to hold all the transferred files, Condor automatically adds the `DiskUsage` job attribute. This attribute includes the total size of the job's executable and all input files to be transferred. Condor then adds an additional clause to the `Requirements` expression that states that the remote machine must have at least enough available disk space to hold all these files:

```
  && (Disk >= DiskUsage)
```

If `should_transfer_files = IF_NEEDED` and the job prefers to run on a machine in the local file system domain over transferring files, (but are still willing to allow the job to run remotely and transfer files), the `rank` expression works well. Use:

```
rank = (TARGET.FileSystemDomain == MY.FileSystemDomain)
```

The `rank` expression is a floating point number, so if other items are considered in ranking the possible machines this job may run on, add the items:

```
rank = kflops + (TARGET.FileSystemDomain == MY.FileSystemDomain)
```

The value of `kflops` can vary widely among machines, so this `rank` expression will likely not do as it intends. To place emphasis on the job running in the same file system domain, but still consider kflops among the machines in the file system domain, weight the part of the rank expression that is matching the file system domains. For example:

```
rank = kflops + (10000 * (TARGET.FileSystemDomain == MY.FileSystemDomain))
```

2.5.5 Environment Variables

The environment under which a job executes often contains information that is potentially useful to the job. Condor allows a user to both set and reference environment variables for a job or job cluster.

Within a submit description file, the user may define environment variables for the job's environment by using the **environment** command. See the *condor_submit* manual page at section 9 for more details about this command.

The submittor's entire environment can be copied into the job ClassAd for the job at job submission. The **getenv** command within the submit description file does this. See the *condor_submit* manual page at section 9 for more details about this command.

If the environment is set with the **environment** command *and* **getenv** is also set to true, values specified with **environment** override values in the submittor's environment (regardless of the order of the **environment** and **getenv** commands).

Commands within the submit description file may reference the environment variables of the submitter as a job is submitted. Submit description file commands use `$ENV(EnvironmentVariableName)` to reference the value of an environment variable. Again, see the *condor_submit* manual page at section 9 for more details about this usage.

Condor sets several additional environment variables for each executing job that may be useful for the job to reference.

- `_CONDOR_SCRATCH_DIR` gives the directory where the job may place temporary data files. This directory is unique for every job that is run, and it's contents are deleted by Condor when the job stops running on a machine, no matter how the job completes.
- `_CONDOR_SLOT` gives the name of the slot (for SMP machines), on which the job is run. On machines with only a single slot, the value of this variable will be `1`, just like the `SlotID` attribute in the machine's ClassAd. This setting is available in all universes. See section 3.13.9 for more details about SMP machines and their configuration.
- `CONDOR_VM` equivalent to `_CONDOR_SLOT` described above, except that it is only available in the standard universe. NOTE: As of Condor version 6.9.3, this environment variable is no longer used. It will only be defined if the `ALLOW_VM_CRUFT` configuration variable is set to `True`.
- `X509_USER_PROXY` gives the full path to the X509 user proxy file if one is associated with the job. (Typically a user will specify **x509userproxy** in the submit file.) This setting is currently available in the local, java, and vanilla universes.

2.5.6 Heterogeneous Submit: Execution on Differing Architectures

If executables are available for the different platforms of machines in the Condor pool, Condor can be allowed the choice of a larger number of machines when allocating a machine for a job. Modifications to the submit description file allow this choice of platforms.

A simplified example is a cross submission. An executable is available for one platform, but the submission is done from a different platform. Given the correct executable, the `requirements` command in the submit description file specifies the target architecture. For example, an executable compiled for a 32-bit Intel processor running Windows Vista, submitted from an Intel architecture running Linux would add the `requirement`

```
requirements = Arch == "INTEL" && OpSys == "WINNT60"
```

Without this `requirement`, *condor_submit* will assume that the program is to be executed on a machine with the same platform as the machine where the job is submitted.

Cross submission works for all universes except `scheduler` and `local`. See section 5.3.10 for how matchmaking works in the `grid` universe. The burden is on the user to both obtain and specify the correct executable for the target architecture. To list the architecture and operating systems of the machines in a pool, run *condor_status*.

Vanilla Universe Example for Execution on Differing Architectures

A more complex example of a heterogeneous submission occurs when a job may be executed on many different architectures to gain full use of a diverse architecture and operating system pool.

If the executables are available for the different architectures, then a modification to the submit description file will allow Condor to choose an executable after an available machine is chosen.

A special-purpose Machine Ad substitution macro can be used in string attributes in the submit description file. The macro has the form

```
$$(MachineAdAttribute)
```

The $$() informs Condor to substitute the requested `MachineAdAttribute` from the machine where the job will be executed.

An example of the heterogeneous job submission has executables available for two platforms: RHEL 3 on both 32-bit and 64-bit Intel processors. This example uses *povray* to render images using a popular free rendering engine.

The substitution macro chooses a specific executable after a platform for running the job is chosen. These executables must therefore be named based on the machine attributes that describe a platform. The executables named

```
povray.LINUX.INTEL
povray.LINUX.X86_64
```

will work correctly for the macro

```
povray.$$(OpSys).$$(Arch)
```

The executables or links to executables with this name are placed into the initial working directory so that they may be found by Condor. A submit description file that queues three jobs for this example:

```
####################
#
# Example of heterogeneous submission
#
####################

universe     = vanilla
Executable   = povray.$$(OpSys).$$(Arch)
Log          = povray.log
Output       = povray.out.$(Process)
Error        = povray.err.$(Process)

Requirements = (Arch == "INTEL" && OpSys == "LINUX") || \
               (Arch == "X86_64" && OpSys =="LINUX")
```

```
Arguments    = +W1024 +H768 +Iimage1.pov
Queue

Arguments    = +W1024 +H768 +Iimage2.pov
Queue

Arguments    = +W1024 +H768 +Iimage3.pov
Queue
```

These jobs are submitted to the vanilla universe to assure that once a job is started on a specific platform, it will finish running on that platform. Switching platforms in the middle of job execution cannot work correctly.

There are two common errors made with the substitution macro. The first is the use of a non-existent `MachineAdAttribute`. If the specified `MachineAdAttribute` does not exist in the machine's ClassAd, then Condor will place the job in the held state until the problem is resolved.

The second common error occurs due to an incomplete job set up. For example, the submit description file given above specifies three available executables. If one is missing, Condor reports back that an executable is missing when it happens to match the job with a resource that requires the missing binary.

Standard Universe Example for Execution on Differing Architectures

Jobs submitted to the standard universe may produce checkpoints. A checkpoint can then be used to start up and continue execution of a partially completed job. For a partially completed job, the checkpoint and the job are specific to a platform. If migrated to a different machine, correct execution requires that the platform must remain the same.

In previous versions of Condor, the author of the heterogeneous submission file would need to write extra policy expressions in the `requirements` expression to force Condor to choose the same type of platform when continuing a checkpointed job. However, since it is needed in the common case, this additional policy is now automatically added to the `requirements` expression. The additional expression is added provided the user does not use `CkptArch` in the `requirements` expression. Condor will remain backward compatible for those users who have explicitly specified `CkptRequirements`–implying use of `CkptArch`, in their `requirements` expression.

The expression added when the attribute `CkptArch` is not specified will default to

```
# Added by Condor
CkptRequirements = ((CkptArch == Arch) || (CkptArch =?= UNDEFINED)) && \
                   ((CkptOpSys == OpSys) || (CkptOpSys =?= UNDEFINED))

Requirements = (<user specified policy>) && $(CkptRequirements)
```

The behavior of the `CkptRequirements` expressions and its addition to `requirements` is as follows. The `CkptRequirements` expression guarantees correct operation in the two possible

cases for a job. In the first case, the job has not produced a checkpoint. The ClassAd attributes `CkptArch` and `CkptOpSys` will be undefined, and therefore the meta operator (`=?=`) evaluates to true. In the second case, the job has produced a checkpoint. The Machine ClassAd is restricted to require further execution only on a machine of the same platform. The attributes `CkptArch` and `CkptOpSys` will be defined, ensuring that the platform chosen for further execution will be the same as the one used just before the checkpoint.

Note that this restriction of platforms also applies to platforms where the executables are binary compatible.

The complete submit description file for this example:

```
####################
#
# Example of heterogeneous submission
#
####################

universe     = standard
Executable   = povray.$$(OpSys).$$(Arch)
Log          = povray.log
Output       = povray.out.$(Process)
Error        = povray.err.$(Process)

# Condor automatically adds the correct expressions to insure that the
# checkpointed jobs will restart on the correct platform types.
Requirements = ( (Arch == "INTEL" && OpSys == "LINUX") || \
               (Arch == "INTEL" && OpSys =="SOLARIS26") || \
               (Arch == "SUN4u" && OpSys == "SOLARIS28") )

Arguments    = +W1024 +H768 +Iimage1.pov
Queue

Arguments    = +W1024 +H768 +Iimage2.pov
Queue

Arguments    = +W1024 +H768 +Iimage3.pov
Queue
```

2.6 Managing a Job

This section provides a brief summary of what can be done once jobs are submitted. The basic mechanisms for monitoring a job are introduced, but the commands are discussed briefly. You are encouraged to look at the man pages of the commands referred to (located in Chapter 9 beginning

on page 690) for more information.

When jobs are submitted, Condor will attempt to find resources to run the jobs. A list of all those with jobs submitted may be obtained through *condor_status* with the *-submitters* option. An example of this would yield output similar to:

```
%  condor_status -submitters

Name                Machine      Running IdleJobs HeldJobs

ballard@cs.wisc.edu  bluebird.c         0       11        0
nice-user.condor@cs. cardinal.c         6      504        0
wright@cs.wisc.edu   finch.cs.w         1        1        0
jbasney@cs.wisc.edu  perdita.cs         0        0        5

                           RunningJobs           IdleJobs           HeldJobs

 ballard@cs.wisc.edu                 0                 11                  0
 jbasney@cs.wisc.edu                 0                  0                  5
nice-user.condor@cs.                 6                504                  0
  wright@cs.wisc.edu                 1                  1                  0

               Total                 7                516                  5
```

2.6.1 Checking on the progress of jobs

At any time, you can check on the status of your jobs with the *condor_q* command. This command displays the status of all queued jobs. An example of the output from *condor_q* is

```
%  condor_q

-- Submitter: submit.chtc.wisc.edu : <128.104.55.9:32772> : submit.chtc.wisc.edu
 ID      OWNER            SUBMITTED     RUN_TIME ST PRI SIZE CMD
711197.0   aragorn         1/15 19:18   0+04:29:33 H  0   0.0  script.sh
894381.0   frodo           3/16 09:06  82+17:08:51 R  0   439.5 elk elk.in
894386.0   frodo           3/16 09:06  82+20:21:28 R  0   219.7 elk elk.in
894388.0   frodo           3/16 09:06  81+17:22:10 R  0   439.5 elk elk.in
1086870.0   gollum          4/27 09:07   0+00:10:14 I  0   7.3  condor_dagman
1086874.0   gollum          4/27 09:08   0+00:00:01 H  0   0.0  RunDC.bat
1297254.0   legolas         5/31 18:05  14+17:40:01 R  0   7.3  condor_dagman
1297255.0   legolas         5/31 18:05  14+17:39:55 R  0   7.3  condor_dagman
1297256.0   legolas         5/31 18:05  14+17:39:55 R  0   7.3  condor_dagman
1297259.0   legolas         5/31 18:05  14+17:39:55 R  0   7.3  condor_dagman
1297261.0   legolas         5/31 18:05  14+17:39:55 R  0   7.3  condor_dagman
1302278.0   legolas         6/4  12:22   1+00:05:37 I  0   390.6 mdrun_1.sh
1304740.0   legolas         6/5  00:14   1+00:03:43 I  0   390.6 mdrun_1.sh
1304967.0   legolas         6/5  05:08   0+00:00:00 I  0   0.0  mdrun_1.sh

14 jobs; 4 idle, 8 running, 2 held
```

This output contains many columns of information about the queued jobs. The ST column (for status) shows the status of current jobs in the queue. An R in the status column means the the job

is currently running. An `I` stands for idle. The job is not running right now, because it is waiting for a machine to become available. The status `H` is the hold state. In the hold state, the job will not be scheduled to run until it is released (see the *condor_hold* reference page located on page 749 and the *condor_release* reference page located on page 786). The `RUN_TIME` time reported for a job is the time that has been committed to the job.

Another useful method of tracking the progress of jobs is through the user log. If you have specified a `log` command in your submit file, the progress of the job may be followed by viewing the log file. Various events such as execution commencement, checkpoint, eviction and termination are logged in the file. Also logged is the time at which the event occurred.

When a job begins to run, Condor starts up a *condor_shadow* process on the submit machine. The shadow process is the mechanism by which the remotely executing jobs can access the environment from which it was submitted, such as input and output files.

It is normal for a machine which has submitted hundreds of jobs to have hundreds of *condor_shadow* processes running on the machine. Since the text segments of all these processes is the same, the load on the submit machine is usually not significant. If there is degraded performance, limit the number of jobs that can run simultaneously by reducing the `MAX_JOBS_RUNNING` configuration variable.

You can also find all the machines that are running your job through the *condor_status* command. For example, to find all the machines that are running jobs submitted by `breach@cs.wisc.edu`, type:

```
%  condor_status -constraint 'RemoteUser == "breach@cs.wisc.edu"'

Name       Arch     OpSys        State      Activity   LoadAv Mem  ActvtyTime

alfred.cs. INTEL    SOLARIS251   Claimed    Busy       0.980  64    0+07:10:02
biron.cs.w INTEL    SOLARIS251   Claimed    Busy       1.000  128   0+01:10:00
cambridge. INTEL    SOLARIS251   Claimed    Busy       0.988  64    0+00:15:00
falcons.cs INTEL    SOLARIS251   Claimed    Busy       0.996  32    0+02:05:03
happy.cs.w INTEL    SOLARIS251   Claimed    Busy       0.988  128   0+03:05:00
istat03.st INTEL    SOLARIS251   Claimed    Busy       0.883  64    0+06:45:01
istat04.st INTEL    SOLARIS251   Claimed    Busy       0.988  64    0+00:10:00
istat09.st INTEL    SOLARIS251   Claimed    Busy       0.301  64    0+03:45:00
...
```

To find all the machines that are running any job at all, type:

```
%  condor_status -run

Name       Arch     OpSys        LoadAv RemoteUser           ClientMachine

adriana.cs INTEL    SOLARIS251   0.980  hepcon@cs.wisc.edu   chevre.cs.wisc.
alfred.cs. INTEL    SOLARIS251   0.980  breach@cs.wisc.edu   neufchatel.cs.w
amul.cs.wi SUN4u    SOLARIS251   1.000  nice-user.condor@cs. chevre.cs.wisc.
anfrom.cs. SUN4x    SOLARIS251   1.023  ashoks@jules.ncsa.ui jules.ncsa.uiuc
anthrax.cs INTEL    SOLARIS251   0.285  hepcon@cs.wisc.edu   chevre.cs.wisc.
astro.cs.w INTEL    SOLARIS251   1.000  nice-user.condor@cs. chevre.cs.wisc.
aura.cs.wi SUN4u    SOLARIS251   0.996  nice-user.condor@cs. chevre.cs.wisc.
balder.cs. INTEL    SOLARIS251   1.000  nice-user.condor@cs. chevre.cs.wisc.
```

```
bamba.cs.w INTEL    SOLARIS251   1.574  dmarino@cs.wisc.edu  riola.cs.wisc.e
bardolph.c INTEL    SOLARIS251   1.000  nice-user.condor@cs. chevre.cs.wisc.
...
```

2.6.2 Removing a job from the queue

A job can be removed from the queue at any time by using the *condor_rm* command. If the job that is being removed is currently running, the job is killed without a checkpoint, and its queue entry is removed. The following example shows the queue of jobs before and after a job is removed.

```
%  condor_q

-- Submitter: froth.cs.wisc.edu : <128.105.73.44:33847> : froth.cs.wisc.edu
 ID      OWNER            SUBMITTED    CPU_USAGE ST PRI SIZE CMD
 125.0   jbasney         4/10 15:35   0+00:00:00 I  -10 1.2  hello.remote
 132.0   raman           4/11 16:57   0+00:00:00 R  0   1.4  hello

2 jobs; 1 idle, 1 running, 0 held

%  condor_rm 132.0
Job 132.0 removed.

%  condor_q

-- Submitter: froth.cs.wisc.edu : <128.105.73.44:33847> : froth.cs.wisc.edu
 ID      OWNER            SUBMITTED    CPU_USAGE ST PRI SIZE CMD
 125.0   jbasney         4/10 15:35   0+00:00:00 I  -10 1.2  hello.remote

1 jobs; 1 idle, 0 running, 0 held
```

2.6.3 Placing a job on hold

A job in the queue may be placed on hold by running the command *condor_hold*. A job in the hold state remains in the hold state until later released for execution by the command *condor_release*.

Use of the *condor_hold* command causes a hard kill signal to be sent to a currently running job (one in the running state). For a standard universe job, this means that no checkpoint is generated before the job stops running and enters the hold state. When released, this standard universe job continues its execution using the most recent checkpoint available.

Jobs in universes other than the standard universe that are running when placed on hold will start over from the beginning when released.

The manual page for *condor_hold* on page 749 and the manual page for *condor_release* on page 786 contain usage details.

2.6.4 Changing the priority of jobs

In addition to the priorities assigned to each user, Condor also provides each user with the capability of assigning priorities to each submitted job. These job priorities are local to each queue and can be any integer value, with higher values meaning better priority.

The default priority of a job is 0, but can be changed using the *condor_prio* command. For example, to change the priority of a job to -15,

```
%  condor_q raman

-- Submitter: froth.cs.wisc.edu : <128.105.73.44:33847> : froth.cs.wisc.edu
 ID      OWNER            SUBMITTED    CPU_USAGE ST PRI SIZE CMD
 126.0   raman           4/11 15:06   0+00:00:00 I  0   0.3  hello

1 jobs; 1 idle, 0 running, 0 held

%  condor_prio -p -15 126.0

%  condor_q raman

-- Submitter: froth.cs.wisc.edu : <128.105.73.44:33847> : froth.cs.wisc.edu
 ID      OWNER            SUBMITTED    CPU_USAGE ST PRI SIZE CMD
 126.0   raman           4/11 15:06   0+00:00:00 I  -15 0.3  hello

1 jobs; 1 idle, 0 running, 0 held
```

It is important to note that these *job* priorities are completely different from the *user* priorities assigned by Condor. Job priorities do not impact user priorities. They are only a mechanism for the user to identify the relative importance of jobs among all the jobs submitted by the user to that specific queue.

2.6.5 Why is the job not running?

Users occasionally find that their jobs do not run. There are many possible reasons why a specific job is not running. The following prose attempts to identify some of the potential issues behind why a job is not running.

At the most basic level, the user knows the status of a job by using *condor_q* to see that the job is not running. By far, the most common reason (to the novice Condor job submitter) why the job is not running is that Condor has not yet been through its periodic negotiation cycle, in which queued jobs are assigned to machines within the pool and begin their execution. This periodic event occurs by default once every 5 minutes, implying that the user ought to wait a few minutes before searching for reasons why the job is not running.

Further inquiries are dependent on whether the job has never run at all, or has run for at least a little bit.

For jobs that have never run, many problems can be diagnosed by using the **-analyze** option of the *condor_q* command. For example, a job (assigned the cluster.process value of 121.000)

submitted to the local pool at UW-Madison is not running. Running *condor_q*'s analyzer provided the following information:

```
% condor_q -pool -analyze 121.000
-- Submitter: puffin.cs.wisc.edu : <128.105.185.14:34203> : puffin.cs.wisc.edu
---
121.000:  Run analysis summary.  Of 1592 machines,
   1382 are rejected by your job's requirements
     25 reject your job because of their own requirements
    185 match but are serving users with a better priority in the pool
      0 match but reject the job for unknown reasons
      0 match but will not currently preempt their existing job
      0 match but are currently offline
      0 are available to run your job

The Requirements expression for your job is:
( ( target.Arch == "X86_64" || target.Arch == "INTEL" ) &&
( target.Group == "TestPool" ) ) && ( target.OpSys == "LINUX" ) &&
( target.Disk >= DiskUsage ) && ( ( target.Memory * 1024 ) >= ImageSize ) &&
( TARGET.FileSystemDomain == MY.FileSystemDomain )

    Condition                         Machines Matched    Suggestion
    ---------                         ----------------    ----------
1   ( target.Group == "TestPool" )    274
2   ( TARGET.FileSystemDomain == "cs.wisc.edu" )1258
3   ( target.OpSys == "LINUX" )       1453
4   ( target.Arch == "X86_64" || target.Arch == "INTEL" )
                                      1573
5   ( target.Disk >= 100000 )         1589
6   ( ( 1024 * target.Memory ) >= 100000 )1592

The following attributes are missing from the job ClassAd:

CheckpointPlatform
```

This example also shows that the job does not run because the job does not have a high enough priority to cause any of 185 other running jobs to be preempted.

While the analyzer can diagnose most common problems, there are some situations that it cannot reliably detect due to the instantaneous and local nature of the information it uses to detect the problem. Thus, it may be that the analyzer reports that resources are available to service the request, but the job still has not run. In most of these situations, the delay is transient, and the job will run following the next negotiation cycle.

A second class of problems represents jobs that do or did run, for at least a short while, but are no longer running. The first issue is identifying whether the job is in this category. The *condor_q* command is not enough; it only tells the current state of the job. The needed information will be in the **log** file or the **error** file, as defined in the submit description file for the job. If these files are not defined, then there is little hope of determining if the job ran at all. For a job that ran, even for the briefest amount of time, the **log** file will contain an event of type 1, which will contain the string `Job executing on host`.

A job may run for a short time, before failing due to a file permission problem. The log file used by the *condor_shadow* daemon will contain more information if this is the problem. This log file is associated with the machine on which the job was submitted. The location and name of this log file may be discovered on the submitting machine, using the command

```
%  condor_config_val SHADOW_LOG
```

Memory and swap space problems may be identified by looking at the log file used by the *condor_schedd* daemon. The location and name of this log file may be discovered on the submitting machine, using the command

```
%  condor_config_val SCHEDD_LOG
```

A swap space problem will show in the log with the following message:

```
2/3 17:46:53 Swap space estimate reached! No more jobs can be run!
12/3 17:46:53     Solution: get more swap space, or set RESERVED_SWAP = 0
12/3 17:46:53     0 jobs matched, 1 jobs idle
```

As an explanation, Condor computes the total swap space on the submit machine. It then tries to limit the total number of jobs it will spawn based on an estimate of the size of the *condor_shadow* daemon's memory footprint and a configurable amount of swap space that should be reserved. This is done to avoid the situation within a very large pool in which all the jobs are submitted from a single host. The huge number of *condor_shadow* processes would overwhelm the submit machine, and it would run out of swap space and thrash.

Things can go wrong if a machine has a lot of physical memory and little or no swap space. Condor does not consider the physical memory size, so the situation occurs where Condor thinks it has no swap space to work with, and it will not run the submitted jobs.

To see how much swap space Condor thinks a given machine has, use the output of a *condor_status* command of the following form:

```
% condor_status -schedd [hostname] -long | grep VirtualMemory
```

If the value listed is 0, then this is what is confusing Condor. There are two ways to fix the problem:

1. Configure the machine with some real swap space.
2. Disable this check within Condor. Define the amount of reserved swap space for the submit machine to 0. Set `RESERVED_SWAP` to 0 in the configuration file:

   ```
   RESERVED_SWAP = 0
   ```

   and then send a *condor_restart* to the submit machine.

2.6.6 In the log file

In a job's log file are a log of events (a listing of events in chronological order) that occurred during the life of the job. The formatting of the events is always the same, so that they may be machine readable. Four fields are always present, and they will most often be followed by other fields that give further information that is specific to the type of event.

The first field in an event is the numeric value assigned as the event type in a 3-digit format. The second field identifies the job which generated the event. Within parentheses are the ClassAd job attributes of `ClusterId` value, `ProcId` value, and the MPI-specific rank for MPI universe jobs or a set of zeros (for jobs run under universes other than MPI), separated by periods. The third field is the date and time of the event logging. The fourth field is a string that briefly describes the event. Fields that follow the fourth field give further information for the specific event type.

These are all of the events that can show up in a job log file:

Event Number: 000
Event Name: Job submitted
Event Description: This event occurs when a user submits a job. It is the first event you will see for a job, and it should only occur once.

Event Number: 001
Event Name: Job executing
Event Description: This shows up when a job is running. It might occur more than once.

Event Number: 002
Event Name: Error in executable
Event Description: The job couldn't be run because the executable was bad.

Event Number: 003
Event Name: Job was checkpointed
Event Description: The job's complete state was written to a checkpoint file. This might happen without the job being removed from a machine, because the checkpointing can happen periodically.

Event Number: 004
Event Name: Job evicted from machine
Event Description: A job was removed from a machine before it finished, usually for a policy reason: perhaps an interactive user has claimed the computer, or perhaps another job is higher priority.

Event Number: 005
Event Name: Job terminated
Event Description: The job has completed.

Event Number: 006
Event Name: Image size of job updated
Event Description: This is informational. It is referring to the memory that the job is using while running. It does not reflect the state of the job.

Event Number: 007
Event Name: Shadow exception
Event Description: The *condor_shadow*, a program on the submit computer that watches over the job and performs some services for the job, failed for some catastrophic reason. The job will leave the machine and go back into the queue.

Event Number: 008
Event Name: Generic log event
Event Description: Not used.

Event Number: 009
Event Name: Job aborted
Event Description: The user canceled the job.

Event Number: 010
Event Name: Job was suspended
Event Description: The job is still on the computer, but it is no longer executing. This is usually for a policy reason, like an interactive user using the computer.

Event Number: 011
Event Name: Job was unsuspended
Event Description: The job has resumed execution, after being suspended earlier.

Event Number: 012
Event Name: Job was held
Event Description: The user has paused the job, perhaps with the *condor_hold* command. It was stopped, and will go back into the queue again until it is aborted or released.

Event Number: 013
Event Name: Job was released
Event Description: The user is requesting that a job on hold be re-run.

Event Number: 014
Event Name: Parallel node executed
Event Description: A parallel (MPI) program is running on a node.

Event Number: 015
Event Name: Parallel node terminated
Event Description: A parallel (MPI) program has completed on a node.

Event Number: 016
Event Name: POST script terminated
Event Description: A node in a DAGMan work flow has a script that should be run after a job. The script is run on the submit host. This event signals that the post script has completed.

Event Number: 017
Event Name: Job submitted to Globus
Event Description: A grid job has been delegated to Globus (version 2, 3, or 4).

Event Number: 018
Event Name: Globus submit failed
Event Description: The attempt to delegate a job to Globus failed.

Event Number: 019
Event Name: Globus resource up
Event Description: The Globus resource that a job wants to run on was unavailable, but is now available.

Event Number: 020
Event Name: Detected Down Globus Resource
Event Description: The Globus resource that a job wants to run on has become unavailable.

Event Number: 021
Event Name: Remote error
Event Description: The *condor_starter* (which monitors the job on the execution machine) has failed.

Event Number: 022
Event Name: Remote system call socket lost
Event Description: The *condor_shadow* and *condor_starter* (which communicate while the job runs) have lost contact.

Event Number: 023
Event Name: Remote system call socket reestablished
Event Description: The *condor_shadow* and *condor_starter* (which communicate while the job runs) have been able to resume contact before the job lease expired.

Event Number: 024
Event Name: Remote system call reconnect failure
Event Description: The *condor_shadow* and *condor_starter* (which communicate while the job runs) were unable to resume contact before the job lease expired.

Event Number: 025
Event Name: Grid Resource Back Up
Event Description: A grid resource that was previously unavailable is now available.

Event Number: 026
Event Name: Detected Down Grid Resource
Event Description: The grid resource that a job is to run on is unavailable.

Event Number: 027
Event Name: Job submitted to grid resource
Event Description: A job has been submitted, and is under the auspices of the grid resource.

Event Number: 028
Event Name: Job ad information event triggered.
Event Description: Extra job ad attributes. This event is written as a supplement to other events when the configuration parameter `EVENT_LOG_JOB_AD_INFORMATION_ATTRS` is set.

Event Number: 029
Event Name: The job's remote status is unknown
Event Description: No updates of the job's remote status have been received for 15 minutes.

Event Number: 030
Event Name: The job's remote status is known again
Event Description: An update has been received for a job whose remote status was previous logged as unknown.

2.6.7 Job Completion

When your Condor job completes(either through normal means or abnormal termination by signal), Condor will remove it from the job queue (i.e., it will no longer appear in the output of *condor_q*) and insert it into the job history file. You can examine the job history file with the *condor_history* command. If you specified a log file in your submit description file, then the job exit status will be recorded there as well.

By default, Condor will send you an email message when your job completes. You can modify this behavior with the *condor_submit* "notification" command. The message will include the exit status of your job (i.e., the argument your job passed to the exit system call when it completed) or notification that your job was killed by a signal. It will also include the following statistics (as appropriate) about your job:

Submitted at: when the job was submitted with *condor_submit*

Completed at: when the job completed

Real Time: elapsed time between when the job was submitted and when it completed (days hours:minutes:seconds)

Run Time: total time the job was running (i.e., real time minus queuing time)

Committed Time: total run time that contributed to job completion (i.e., run time minus the run time that was lost because the job was evicted without performing a checkpoint)

Remote User Time: total amount of committed time the job spent executing in user mode

Remote System Time: total amount of committed time the job spent executing in system mode

Total Remote Time: total committed CPU time for the job

Local User Time: total amount of time this job's *condor_shadow* (remote system call server) spent executing in user mode

Local System Time: total amount of time this job's *condor_shadow* spent executing in system mode

Total Local Time: total CPU usage for this job's *condor_shadow*

Leveraging Factor: the ratio of total remote time to total system time (a factor below 1.0 indicates that the job ran inefficiently, spending more CPU time performing remote system calls than actually executing on the remote machine)

Virtual Image Size: memory size of the job, computed when the job checkpoints

Checkpoints written: number of successful checkpoints performed by the job

Checkpoint restarts: number of times the job successfully restarted from a checkpoint

Network: total network usage by the job for checkpointing and remote system calls

Buffer Configuration: configuration of remote system call I/O buffers

Total I/O: total file I/O detected by the remote system call library

I/O by File: I/O statistics per file produced by the remote system call library

Remote System Calls: listing of all remote system calls performed (both Condor-specific and Unix system calls) with a count of the number of times each was performed

2.7 Priorities and Preemption

Condor has two independent priority controls: *job* priorities and *user* priorities.

2.7.1 Job Priority

Job priorities allow the assignment of a priority level to each submitted Condor job in order to control order of execution. To set a job priority, use the *condor_prio* command — see the example in section 2.6.4, or the command reference page on page 767. Job priorities do not impact user priorities in any fashion. A job priority can be any integer, and higher values are “better”.

2.7.2 User priority

Machines are allocated to users based upon a user’s priority. A lower numerical value for user priority means higher priority, so a user with priority 5 will get more resources than a user with priority 50. User priorities in Condor can be examined with the *condor_userprio* command (see page 873). Condor administrators can set and change individual user priorities with the same utility.

Condor continuously calculates the share of available machines that each user should be allocated. This share is inversely related to the ratio between user priorities. For example, a user with a priority of 10 will get twice as many machines as a user with a priority of 20. The priority of each individual user changes according to the number of resources the individual is using. Each user starts out with the best possible priority: 0.5. If the number of machines a user currently has is

greater than the user priority, the user priority will worsen by numerically increasing over time. If the number of machines is less then the priority, the priority will improve by numerically decreasing over time. The long-term result is fair-share access across all users. The speed at which Condor adjusts the priorities is controlled with the configuration macro `PRIORITY_HALFLIFE`, an exponential half-life value. The default is one day. If a user that has user priority of 100 and is utilizing 100 machines removes all his/her jobs, one day later that user's priority will be 50, and two days later the priority will be 25.

Condor enforces that each user gets his/her fair share of machines according to user priority both when allocating machines which become available and by priority preemption of currently allocated machines. For instance, if a low priority user is utilizing all available machines and suddenly a higher priority user submits jobs, Condor will immediately checkpoint and vacate jobs belonging to the lower priority user. This will free up machines that Condor will then give over to the higher priority user. Condor will not starve the lower priority user; it will preempt only enough jobs so that the higher priority user's fair share can be realized (based upon the ratio between user priorities). To prevent thrashing of the system due to priority preemption, the Condor site administrator can define a `PREEMPTION_REQUIREMENTS` expression in Condor's configuration. The default expression that ships with Condor is configured to only preempt lower priority jobs that have run for at least one hour. So in the previous example, in the worse case it could take up to a maximum of one hour until the higher priority user receives his fair share of machines. For a general discussion of limiting preemption, please see section 3.5.9 of the Administrator's manual.

User priorities are keyed on "username@domain", for example "johndoe@cs.wisc.edu". The domain name to use, if any, is configured by the Condor site administrator. Thus, user priority and therefore resource allocation is not impacted by which machine the user submits from or even if the user submits jobs from multiple machines.

An extra feature is the ability to submit a job as a *nice* job (see page 852). Nice jobs artificially boost the user priority by one million just for the nice job. This effectively means that nice jobs will only run on machines that no other Condor job (that is, non-niced job) wants. In a similar fashion, a Condor administrator could set the user priority of any specific Condor user very high. If done, for example, with a guest account, the guest could only use cycles not wanted by other users of the system.

2.7.3 Details About How Condor Jobs Vacate Machines

When Condor needs a job to vacate a machine for whatever reason, it sends the job an asynchronous signal specified in the `KillSig` attribute of the job's ClassAd. The value of this attribute can be specified by the user at submit time by placing the **kill_sig** option in the Condor submit description file.

If a program wanted to do some special work when required to vacate a machine, the program may set up a signal handler to use a trappable signal as an indication to clean up. When submitting this job, this clean up signal is specified to be used with **kill_sig**. Note that the clean up work needs to be quick. If the job takes too long to go away, Condor follows up with a SIGKILL signal which immediately terminates the process.

A job that is linked using *condor_compile* and is subsequently submitted into the standard universe, will checkpoint and exit upon receipt of a SIGTSTP signal. Thus, SIGTSTP is the default value for `KillSig` when submitting to the standard universe. The user's code may still checkpoint itself at any time by calling one of the following functions exported by the Condor libraries:

`ckpt()()` Performs a checkpoint and then returns.

`ckpt_and_exit()()` Checkpoints and exits; Condor will then restart the process again later, potentially on a different machine.

For jobs submitted into the vanilla universe, the default value for `KillSig` is SIGTERM, the usual method to nicely terminate a Unix program.

2.8 Java Applications

Condor allows users to access a wide variety of machines distributed around the world. The Java Virtual Machine (JVM) provides a uniform platform on any machine, regardless of the machine's architecture or operating system. The Condor Java universe brings together these two features to create a distributed, homogeneous computing environment.

Compiled Java programs can be submitted to Condor, and Condor can execute the programs on any machine in the pool that will run the Java Virtual Machine.

The *condor_status* command can be used to see a list of machines in the pool for which Condor can use the Java Virtual Machine.

```
% condor_status -java

Name               JavaVendor Ver    State     Activity LoadAv  Mem  ActvtyTime

adelie01.cs.wisc.e Sun Micros 1.6.0_ Claimed   Busy     0.090   873  0+00:02:46
adelie02.cs.wisc.e Sun Micros 1.6.0_ Owner     Idle     0.210   873  0+03:19:32
slot10@bio.cs.wisc Sun Micros 1.6.0_ Unclaimed Idle     0.000   118  7+03:13:28
slot2@bio.cs.wisc. Sun Micros 1.6.0_ Unclaimed Idle     0.000   118  7+03:13:28
...
```

If there is no output from the *condor_status* command, then Condor does not know the location details of the Java Virtual Machine on machines in the pool, or no machines have Java correctly installed. In this case, contact your system administrator or see section 3.14 for more information on getting Condor to work together with Java.

2.8.1 A Simple Example Java Application

Here is a complete, if simple, example. Start with a simple Java program, `Hello.java`:

```
public class Hello {
        public static void main( String [] args ) {
                System.out.println("Hello, world!\n");
        }
}
```

Build this program using your Java compiler. On most platforms, this is accomplished with the command

```
javac Hello.java
```

Submission to Condor requires a submit description file. If submitting where files are accessible using a shared file system, this simple submit description file works:

```
####################
#
# Example 1
# Execute a single Java class
#
####################

universe       = java
executable     = Hello.class
arguments      = Hello
output         = Hello.output
error          = Hello.error
queue
```

The Java universe must be explicitly selected.

The main class of the program is given in the **executable** statement. This is a file name which contains the entry point of the program. The name of the main class (not a file name) must be specified as the first argument to the program.

If submitting the job where a shared file system is *not* accessible, the submit description file becomes:

```
####################
#
# Example 1
# Execute a single Java class,
# not on a shared file system
#
####################
```

```
universe       = java
executable     = Hello.class
arguments      = Hello
output         = Hello.output
error          = Hello.error
should_transfer_files = YES
when_to_transfer_output = ON_EXIT
queue
```

For more information about using Condor's file transfer mechanisms, see section 2.5.4.

To submit the job, where the submit description file is named `Hello.cmd`, execute

```
condor_submit Hello.cmd
```

To monitor the job, the commands *condor_q* and *condor_rm* are used as with all jobs.

2.8.2 Less Simple Java Specifications

Specifying more than 1 class file. For programs that consist of more than one `.class` file, identify the files in the submit description file:

```
executable = Stooges.class
transfer_input_files = Larry.class,Curly.class,Moe.class
```

The **executable** command does not change. It still identifies the class file that contains the program's entry point.

JAR files. If the program consists of a large number of class files, it may be easier to collect them all together into a single Java Archive (JAR) file. A JAR can be created with:

```
% jar cvf Library.jar Larry.class Curly.class Moe.class Stooges.class
```

Condor must then be told where to find the JAR as well as to use the JAR. The JAR file that contains the entry point is specified with the **executable** command. All JAR files are specified with the **jar_files** command. For this example that collected all the class files into a single JAR file, the submit description file contains:

```
executable = Library.jar
jar_files = Library.jar
```

Note that the JVM must know whether it is receiving JAR files or class files. Therefore, Condor must also be informed, in order to pass the information on to the JVM. That is why

there is a difference in submit description file commands for the two ways of specifying files (**transfer_input_files** and **jar_files**).

If there are multiple JAR files, the **executable** command specifies the JAR file that contains the program's entry point. This file is also listed with the **jar_files** command:

```
executable = sortmerge.jar
jar_files = sortmerge.jar,statemap.jar
```

Using a third-party JAR file. As Condor requires that all JAR files (third-party or not) be available, specification of a third-party JAR file is no different than other JAR files. If the sortmerge example above also relies on version 2.1 from http://jakarta.apache.org/commons/lang/, and this JAR file has been placed in the same directory with the other JAR files, then the submit description file contains

```
executable = sortmerge.jar
jar_files = sortmerge.jar,statemap.jar,commons-lang-2.1.jar
```

An executable JAR file. When the JAR file is an executable, specify the program's entry point in the **arguments** command:

```
executable = anexecutable.jar
jar_files  = anexecutable.jar
arguments  = some.main.ClassFile
```

Packages. An example of a Java class that is declared in a non-default package is

```
package hpc;

 public class CondorDriver
 {
     // class definition here
 }
```

The JVM needs to know the location of this package. It is passed as a command-line argument, implying the use of the naming convention and directory structure.

Therefore, the submit description file for this example will contain

```
arguments = hpc.CondorDriver
```

JVM-version specific features. If the program uses Java features found only in certain JVMs, then the Java application submitted to Condor must only run on those machines within the pool that run the needed JVM. Inform Condor by adding a `requirements` statement to the submit description file. For example, to require version 3.2, add to the submit description file:

```
requirements = (JavaVersion=="3.2")
```

Benchmark speeds. Each machine with Java capability in a Condor pool will execute a benchmark to determine its speed. The benchmark is taken when Condor is started on the machine, and it uses the SciMark2 (http://math.nist.gov/scimark2) benchmark. The result of the benchmark is held as an attribute within the machine ClassAd. The attribute is called `JavaMFlops`. Jobs that are run under the Java universe (as all other Condor jobs) may prefer or require a machine of a specific speed by setting `rank` or `requirements` in the submit description file. As an example, to execute only on machines of a minimum speed:

```
requirements = (JavaMFlops>4.5)
```

JVM options. Options to the JVM itself are specified in the submit description file:

```
java_vm_args = -DMyProperty=Value -verbose:gc -Xmx1024m
```

These options are those which go after the java command, but before the user's main class. Do not use this to set the classpath, as Condor handles that itself. Setting these options is useful for setting system properties, system assertions and debugging certain kinds of problems.

2.8.3 Chirp I/O

If a job has more sophisticated I/O requirements that cannot be met by Condor's file transfer mechanism, then the Chirp facility may provide a solution. Chirp has two advantages over simple, whole-file transfers. First, it permits the input files to be decided upon at run-time rather than submit time, and second, it permits partial-file I/O with results than can be seen as the program executes. However, small changes to the program are required in order to take advantage of Chirp. Depending on the style of the program, use either Chirp I/O streams or UNIX-like I/O functions.

Chirp I/O streams are the easiest way to get started. Modify the program to use the objects `ChirpInputStream` and `ChirpOutputStream` instead of `FileInputStream` and `FileOutputStream`. These classes are completely documented in the Condor Software Developer's Kit (SDK). Here is a simple code example:

```
import java.io.*;
import edu.wisc.cs.condor.chirp.*;

public class TestChirp {

   public static void main( String args[] ) {

      try {
         BufferedReader in = new BufferedReader(
            new InputStreamReader(
               new ChirpInputStream("input")));
```

```
         PrintWriter out = new PrintWriter(
            new OutputStreamWriter(
               new ChirpOutputStream("output")));

         while(true) {
            String line = in.readLine();
            if(line==null) break;
            out.println(line);
         }
         out.close();
      } catch( IOException e ) {
         System.out.println(e);
      }
   }
}
```

To perform UNIX-like I/O with Chirp, create a `ChirpClient` object. This object supports familiar operations such as `open`, `read`, `write`, and `close`. Exhaustive detail of the methods may be found in the Condor SDK, but here is a brief example:

```
import java.io.*;
import edu.wisc.cs.condor.chirp.*;

public class TestChirp {

   public static void main( String args[] ) {

      try {
         ChirpClient client = new ChirpClient();
         String message = "Hello, world!\n";
         byte [] buffer = message.getBytes();

         // Note that we should check that actual==length.
         // However, skip it for clarity.

         int fd = client.open("output","wct",0777);
         int actual = client.write(fd,buffer,0,buffer.length);
         client.close(fd);

         client.rename("output","output.new");
         client.unlink("output.new");

      } catch( IOException e ) {
         System.out.println(e);
      }
```

```
   }
}
```

Regardless of which I/O style, the Chirp library must be specified and included with the job. The Chirp JAR (`Chirp.jar`) is found in the `lib` directory of the Condor installation. Copy it into your working directory in order to compile the program after modification to use Chirp I/O.

```
% condor_config_val LIB
/usr/local/condor/lib
% cp /usr/local/condor/lib/Chirp.jar .
```

Rebuild the program with the Chirp JAR file in the class path.

```
% javac -classpath Chirp.jar:. TestChirp.java
```

The Chirp JAR file must be specified in the submit description file. Here is an example submit description file that works for both of the given test programs:

```
universe = java
executable = TestChirp.class
arguments = TestChirp
jar_files = Chirp.jar
queue
```

2.9 Parallel Applications (Including MPI Applications)

Condor's Parallel universe supports a wide variety of parallel programming environments, and it encompasses the execution of MPI jobs. It supports jobs which need to be co-scheduled. A co-scheduled job has more than one process that must be running at the same time on different machines to work correctly. The parallel universe supersedes the mpi universe. The mpi universe eventually will be removed from Condor.

2.9.1 Prerequisites to Running Parallel Jobs

Condor must be configured such that resources (machines) running parallel jobs are dedicated. Note that *dedicated* has a very specific meaning in Condor: dedicated machines never vacate their executing Condor jobs, should the machine's interactive owner return. This is implemented by running a single dedicated scheduler process on a machine in the pool, which becomes the single machine from which parallel universe jobs are submitted. Once the dedicated scheduler claims a dedicated machine for use, the dedicated scheduler will try to use that machine to satisfy the requirements of

the queue of parallel universe or MPI universe jobs. If the dedicated scheduler cannot use a machine for a configurable amount of time, it will release its claim on the machine, making it available again for the opportunistic scheduler.

Since Condor does not ordinarily run this way, (Condor usually uses opportunistic scheduling), dedicated machines must be specially configured. Section 3.13.10 of the Administrator's Manual describes the necessary configuration and provides detailed examples.

To simplify the scheduling of dedicated resources, a single machine becomes the scheduler of dedicated resources. This leads to a further restriction that jobs submitted to execute under the parallel universe must be submitted from the machine acting as the dedicated scheduler.

2.9.2 Parallel Job Submission

Given correct configuration, parallel universe jobs may be submitted from the machine running the dedicated scheduler. The dedicated scheduler claims machines for the parallel universe job, and invokes the job when the correct number of machines of the correct platform (architecture and operating system) are claimed. Note that the job likely consists of more than one process, each to be executed on a separate machine. The first process (machine) invoked is treated different than the others. When this first process exits, Condor shuts down all the others, even if they have not yet completed their execution.

An overly simplified submit description file for a parallel universe job appears as

```
############################################
##   submit description file for a parallel program
############################################
universe = parallel
executable = /bin/sleep
arguments = 30
machine_count = 8
queue
```

This job specifies the **universe** as **parallel**, letting Condor know that dedicated resources are required. The **machine_count** command identifies the number of machines required by the job.

When submitted, the dedicated scheduler allocates eight machines with the same architecture and operating system as the submit machine. It waits until all eight machines are available before starting the job. When all the machines are ready, it invokes the */bin/sleep* command, with a command line argument of 30 on all eight machines more or less simultaneously.

A more realistic example of a parallel job utilizes other features.

```
######################################
## Parallel example submit description file
```

```
######################################
universe = parallel
executable = /bin/cat
log = logfile
input = infile.$(NODE)
output = outfile.$(NODE)
error = errfile.$(NODE)
machine_count = 4
queue
```

The specification of the **input**, **output**, and **error** files utilize the predefined macro `$(NODE)`. See the *condor_submit* manual page on page 825 for further description of predefined macros. The `$(NODE)` macro is given a unique value as processes are assigned to machines. The `$(NODE)` value is fixed for the entire length of the job. It can therefore be used to identify individual aspects of the computation. In this example, it is used to utilize and assign unique names to input and output files.

This example presumes a shared file system across all the machines claimed for the parallel universe job. Where no shared file system is either available or guaranteed, use Condor's file transfer mechanism, as described in section 2.5.4 on page 25. This example uses the file transfer mechanism.

```
######################################
## Parallel example submit description file
## without using a shared file system
######################################
universe = parallel
executable = /bin/cat
log = logfile
input = infile.$(NODE)
output = outfile.$(NODE)
error = errfile.$(NODE)
machine_count = 4
should_transfer_files = yes
when_to_transfer_output = on_exit
queue
```

The job requires exactly four machines, and queues four processes. Each of these processes requires a correctly named input file, and produces an output file.

2.9.3 Parallel Jobs with Separate Requirements

The different machines executing for a parallel universe job may specify different machine requirements. A common example requires that the head node execute on a specific machine. It may be also useful for debugging purposes.

Consider the following example.

```
######################################
## Example submit description file
## with multiple procs
######################################
universe = parallel
executable = example
machine_count = 1
requirements = ( machine == "machine1")
queue

requirements = ( machine =!= "machine1")
machine_count = 3
queue
```

The dedicated scheduler allocates four machines. All four executing jobs have the same value for `$(Cluster)` macro. The `$(Process)` macro takes on two values; the value 0 will be assigned for the single executable that must be executed on machine1, and the value 1 will be assigned for the other three that must be executed anywhere but on machine1.

Carefully consider the ordering and nature of multiple sets of requirements in the same submit description file. The scheduler matches jobs to machines based on the ordering within the submit description file. Mutually exclusive requirements eliminate the dependence on ordering within the submit description file. Without mutually exclusive requirements, the scheduler may be unable to schedule the job. The ordering within the submit description file may preclude the scheduler considering the specific allocation that could satisfy the requirements.

2.9.4 MPI Applications Within Condor's Parallel Universe

MPI applications utilize a single executable that is invoked in order to execute in parallel on one or more machines. Condor's parallel universe provides the environment within which this executable is executed in parallel. However, the various implementations of MPI (for example, LAM or MPICH) require further framework items within a system-wide environment. Condor supports this necessary framework through user visible and modifiable scripts. An MPI implementation-dependent script becomes the Condor job. The script sets up the extra, necessary framework, and then invokes the MPI application's executable.

Condor provides these scripts in the `$(RELEASE_DIR)/etc/examples` directory. The script for the LAM implementation is `lamscript`. The script for the MPICH implementation is `mp1script`. Therefore, a Condor submit description file for these implementations would appear similar to:

```
######################################
```

```
## Example submit description file
## for MPICH 1 MPI
## works with MPICH 1.2.4, 1.2.5 and 1.2.6
######################################
universe = parallel
executable = mp1script
arguments = my_mpich_linked_executable arg1 arg2
machine_count = 4
should_transfer_files = yes
when_to_transfer_output = on_exit
transfer_input_files = my_mpich_linked_executable
queue
```

or

```
######################################
## Example submit description file
## for LAM MPI
######################################
universe = parallel
executable = lamscript
arguments = my_lam_linked_executable arg1 arg2
machine_count = 4
should_transfer_files = yes
when_to_transfer_output = on_exit
transfer_input_files = my_lam_linked_executable
queue
```

The **executable** is the MPI implementation-dependent script. The first argument to the script is the MPI application's executable. Further arguments to the script are the MPI application's arguments. Condor must transfer this executable; do this with the **transfer_input_files** command.

For other implementations of MPI, copy and modify one of the given scripts. Most MPI implementations require two system-wide prerequisites. The first prerequisite is the ability to run a command on a remote machine without being prompted for a password. *ssh* is commonly used, but other commands may be used. The second prerequisite is an ASCII file containing the list of machines that may utilize *ssh*. These common prerequisites are implemented in a further script called `sshd.sh`. `sshd.sh` generates ssh keys (to enable password-less remote execution), and starts an *sshd* daemon. The machine name and MPI rank are given to the submit machine.

The *sshd.sh* script requires the definition of two Condor configuration variables. Configuration variable `CONDOR_SSHD` is an absolute path to an implementation of *sshd*. *sshd.sh* has been tested with *openssh* version 3.9, but should work with more recent versions. Configuration variable `CONDOR_SSH_KEYGEN` points to the corresponding *ssh-keygen* executable.

Scripts *lamscript* and *mp1script* each have their own idiosyncrasies. In *mp1script*, the `PATH` to the MPICH installation must be set. The shell variable MPDIR indicates its proper value. This

directory contains the MPICH *mpirun* executable. For LAM, there is a similar path setting, but it is called `LAMDIR` in the *lamscript* script. In addition, this path must be part of the path set in the user's `.cshrc` script. As of this writing, the LAM implementation does not work if the user's login shell is the Bourne or compatible shell.

These MPI jobs operate as all parallel universe jobs do. The default policy is that when the first node exits, the whole job is considered done, and Condor kills all other running nodes in that parallel job. Alternatively, a parallel universe job that sets the attribute

```
+ParallelShutdownPolicy = "WAIT_FOR_ALL"
```

in its submit description file changes the policy, such that Condor will wait until every node in the parallel job has completed to consider the job finished.

2.10 DAGMan Applications

A directed acyclic graph (DAG) can be used to represent a set of computations where the input, output, or execution of one or more computations is dependent on one or more other computations. The computations are nodes (vertices) in the graph, and the edges (arcs) identify the dependencies. Condor finds machines for the execution of programs, but it does not schedule programs based on dependencies. The Directed Acyclic Graph Manager (DAGMan) is a meta-scheduler for the execution of programs (computations). DAGMan submits the programs to Condor in an order represented by a DAG and processes the results. A DAG input file describes the DAG, and further submit description file(s) are used by DAGMan when submitting programs to run under Condor.

DAGMan is itself executed as a scheduler universe job within Condor. As DAGMan submits programs, it monitors log file(s) to enforce the ordering required within the DAG. DAGMan is also responsible for scheduling, recovery, and reporting on the set of programs submitted to Condor.

2.10.1 DAGMan Terminology

To DAGMan, a node in a DAG may encompass more than a single program submitted to run under Condor. Figure 2.2 illustrates the elements of a node.

At one time, the number of Condor jobs per node was restricted to one. This restriction is now relaxed such that all Condor jobs within a node must share a single cluster number. See the *condor_submit* manual page for a further definition of a cluster. A limitation exists such that all jobs within the single cluster must use the same log file. Separate nodes within a DAG may use different log files.

As DAGMan schedules and submits jobs within nodes to Condor, these jobs are defined to succeed or fail based on their return values. This success or failure is propagated in well-defined ways to the level of a node within a DAG. Further progression of computation (towards completing the DAG) may be defined based upon the success or failure of one or more nodes.

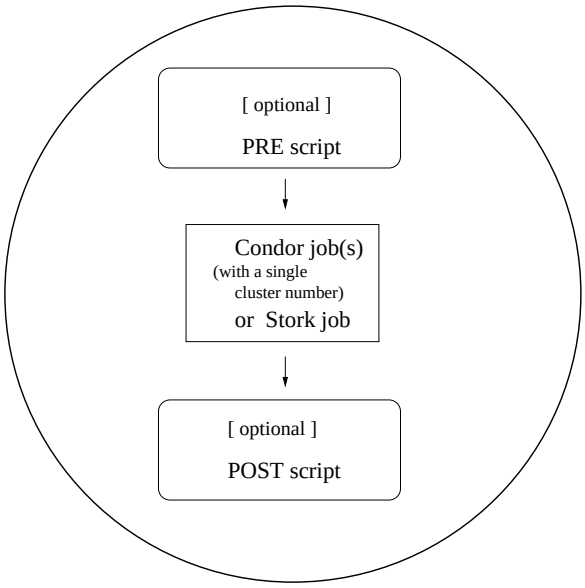

Figure 2.2: One Node within a DAG

The failure of a single job within a cluster of multiple jobs (within a single node) causes the entire cluster of jobs to fail. Any other jobs within the failed cluster of jobs are immediately removed. Each node within a DAG is further defined to succeed or fail, based upon the return values of a PRE script, the job(s) within the cluster, and/or a POST script.

2.10.2 Input File Describing the DAG: the JOB, DATA, SCRIPT and PARENT...CHILD Key Words

The input file used by DAGMan is called a DAG input file. All items are optional, but there must be at least one *JOB* or *DATA* item.

Comments may be placed in the DAG input file. The pound character (#) as the first character on a line identifies the line as a comment. Comments do not span lines.

A simple diamond-shaped DAG, as shown in Figure 2.3 is presented as a starting point for examples. This DAG contains 4 nodes.

A very simple DAG input file for this diamond-shaped DAG is

```
# File name: diamond.dag
#
JOB  A  A.condor
JOB  B  B.condor
JOB  C  C.condor
```

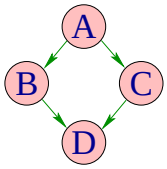

Figure 2.3: Diamond DAG

```
JOB  D  D.condor
PARENT A CHILD B C
PARENT B C CHILD D
```

A set of basic key words appearing in a DAG input file is described below.

- **JOB**

  The *JOB* key word specifies a job to be managed by Condor. The syntax used for each *JOB* entry is

  JOB *JobName SubmitDescriptionFileName* [**DIR** *directory*] [**NOOP**] [**DONE**]

  A *JOB* entry maps a *JobName* to a Condor submit description file. The *JobName* uniquely identifies nodes within the DAGMan input file and in output messages. Note that the name for each node within the DAG must be unique.

  The key words *JOB*, *DIR*, *NOOP*, and *DONE* are not case sensitive. Therefore, *DONE*, *Done*, and *done* are all equivalent. The values defined for *JobName* and *SubmitDescriptionFileName* are case sensitive, as file names in the Unix file system are case sensitive. The *JobName* can be any string that contains no white space, except for the strings *PARENT* and *CHILD* (in upper, lower, or mixed case).

  Note that *DIR*, *NOOP*, and *DONE*, if used, must appear in the order shown above.

  The *DIR* option specifies a working directory for this node, from which the Condor job will be submitted, and from which a *PRE* and/or *POST* script will be run. Note that a DAG containing *DIR* specifications cannot be run in conjunction with the *-usedagdir* command-line argument to *condor_submit_dag*. A rescue DAG generated by a DAG run with the *-usedagdir* argument will contain *DIR* specifications, so the *-usedagdir* argument is automatically disregarded when running a rescue DAG.

  The optional *NOOP* keyword identifies that the Condor job within the node is not to be submitted to Condor. This optimization is useful in cases such as debugging a complex DAG structure, where some of the individual jobs are long-running. For this debugging of structure, some jobs are marked as *NOOP*s, and the DAG is initially run to verify that the control flow through the DAG is correct. The *NOOP* keywords are then removed before submitting the DAG. Any PRE and POST scripts for jobs specified with *NOOP are* executed; to avoid running the PRE and POST scripts, comment them out. The job that is not submitted to Condor is given a return value that indicates success, such that the node may also succeed. Return values of any PRE and POST scripts may still cause the node to fail. Even though the job specified with *NOOP* is not submitted, its submit description file must exist; the log file for

the job is used, because DAGMan generates dummy submission and termination events for the job.

The optional *DONE* keyword identifies a node as being already completed. This is mainly used by rescue DAGs generated by DAGMan itself, in the event of a failure to complete the workflow. Nodes with the *DONE* keyword are not executed when the rescue DAG is run, allowing the workflow to pick up from the previous endpoint. Users should generally not use the *DONE* keyword. The *NOOP* keyword is more flexible in avoiding the execution of a job within a node. Note that, for any node marked *DONE* in a DAG, all of its parents must also be marked *DONE*; otherwise, a fatal error will result. The *DONE* keyword applies to the entire node. A node marked with *DONE* will not have a PRE or POST script run, and the Condor job will not be submitted.

- **DATA**

  The *DATA* key word specifies a job to be managed by the Stork data placement server. Stork software is provided by the Stork project. Please refer to their website: http://www.cct.lsu.edu/ kosar/stork/index.php.

  The syntax used for each *DATA* entry is

  DATA *JobName SubmitDescriptionFileName* [**DIR** *directory*] [**NOOP**] [**DONE**]

  A *DATA* entry maps a *JobName* to a Stork submit description file. In all other respects, the *DATA* key word is identical to the *JOB* key word.

  The keywords *DIR*, *NOOP* and *DONE* follow the same rules and restrictions, and they have the same effect for **DATA** nodes as they do for **JOB** nodes.

  Here is an example of a simple DAG that stages in data using Stork, processes the data using Condor, and stages the processed data out using Stork. Depending upon the implementation, multiple data jobs to stage in data or to stage out data may be run in parallel.

```
DATA    STAGE_IN1  stage_in1.stork
DATA    STAGE_IN2  stage_in2.stork
JOB     PROCESS    process.condor
DATA    STAGE_OUT1 stage_out1.stork
DATA    STAGE_OUT2 stage_out2.stork
PARENT  STAGE_IN1 STAGE_IN2 CHILD PROCESS
PARENT  PROCESS CHILD STAGE_OUT1 STAGE_OUT2
```

- **SCRIPT**

  The *SCRIPT* key word specifies processing that is done either before a job within the DAG is submitted to Condor or Stork for execution or after a job within the DAG completes its execution. Processing done before a job is submitted to Condor or Stork is called a *PRE* script. Processing done after a job completes its execution under Condor or Stork is called a *POST* script. A node in the DAG is comprised of the job together with *PRE* and/or *POST* scripts.

  PRE and *POST* script lines within the DAG input file use the syntax:

  SCRIPT PRE *JobName ExecutableName* [*arguments*]

  SCRIPT POST *JobName ExecutableName* [*arguments*]

The *SCRIPT* key word identifies the type of line within the DAG input file. The *PRE* or *POST* key word specifies the relative timing of when the script is to be run. The *JobName* specifies the node to which the script is attached. The *ExecutableName* specifies the script to be executed, and it may be followed by any command line arguments to that script. The *ExecutableName* and optional *arguments* are case sensitive; they have their case preserved. **Note that neither the *ExecutableName* nor the individual arguments within the *arguments* string can contain spaces.**

Scripts are optional for each job, and any scripts are executed on the machine from which the DAG is submitted; this is not necessarily the same machine upon which the node's Condor or Stork job is run. Further, a single cluster of Condor jobs may be spread across several machines.

A PRE script is commonly used to place files in a staging area for the cluster of jobs to use. A POST script is commonly used to clean up or remove files once the cluster of jobs is finished running. An example uses PRE and POST scripts to stage files that are stored on tape. The PRE script reads compressed input files from the tape drive, and it uncompresses them, placing the input files in the current directory. The cluster of Condor jobs reads these input files and produces output files. The POST script compresses the output files, writes them out to the tape, and then removes both the staged input files and the output files.

DAGMan takes note of the exit value of the scripts as well as the job or jobs within the cluster. A script with an exit value not equal to 0 fails. If the PRE script fails, then neither the job nor the POST script runs, and the node fails.

If the PRE script succeeds, the Condor or Stork job is submitted. If the job or any one of the jobs within the single cluster fails and there is no POST script, the DAG node is marked as failed. An exit value not equal to 0 indicates program failure. It is therefore important that a successful program return the exit value 0.

If the job fails and there is a POST script, node failure is determined by the exit value of the POST script. A failing value from the POST script marks the node as failed. A succeeding value from the POST script (even with a failed job) marks the node as successful. Therefore, the POST script may need to consider the return value from the job.

By default, the POST script is run regardless of the job's return value.

A node not marked as failed at any point is successful. Table 2.1 summarizes the success or failure of an entire node for all possibilities. An *S* stands for success, an *F* stands for failure, and the dash character (-) identifies that there is no script.

PRE	-	-	F	S	S	-	-	-	-	S	S	S	S
JOB	S	F	not run	S	F	S	S	F	F	S	F	F	S
POST	-	-	not run	-	-	S	F	S	F	S	S	F	F
node	S	F	F	S	F	S	F	S	F	S	S	F	F

Table 2.1: Node success or failure definition

Five variables (`$JOB`, `$JOBID`, `$RETRY`, `$MAX_RETRIES`, and `$RETURN`) can be used within the DAG input file as arguments passed to a PRE or POST script.

The variable `$JOB` evaluates to the (case sensitive) string defined for *JobName*.

The variable `$RETRY` evaluates to an integer value set to 0 the first time a node is run, and is incremented each time the node is retried. See section 2.10.6 for the description of how to cause nodes to be retried.

The variable `$MAX_RETRIES` evaluates to an integer value set to the maximum number of retries for the node. See section 2.10.6 for the description of how to cause nodes to be retried. If no retries are set for the node, `$MAX_RETRIES` will be set to 0.

For use as an argument to POST scripts only, the variable `$JOBID` evaluates to a representation of the Condor job ID of the node job. It is the value of the job ClassAd attribute `ClusterId`, followed by a period, and then followed by the value of the job ClassAd attribute `ProcId`. An example of a job ID might be 1234.0. For nodes with multiple jobs in the same cluster, the `ProcId` value is the one of the last job within the cluster.

For use as an argument to POST scripts only, the `$RETURN` variable evaluates to the return value of the Condor or Stork job, if there is a single job within a cluster. With multiple jobs within the same cluster, there are two cases to consider. In the first case, all jobs within the cluster are successful; the value of `$RETURN` will be 0, indicating success. In the second case, one or more jobs from the cluster fail. When *condor_dagman* sees the first terminated event for a job that failed, it assigns that job's return value as the value of `$RETURN`, and attempts to remove all remaining jobs within the cluster. Therefore, if multiple jobs in the cluster fail with different exit codes, a race condition determines which exit code gets assigned to `$RETURN`.

A job that dies due to a signal is reported with a `$RETURN` value representing the additive inverse of the signal number. For example, SIGKILL (signal 9) is reported as -9. A job whose batch system submission fails is reported as -1001. A job that is externally removed from the batch system queue (by something other than *condor_dagman*) is reported as -1002.

As an example, consider the diamond-shaped DAG example. Suppose the PRE script expands a compressed file needed as input to nodes B and C. The file is named of the form *`JobName`*`.gz`. The DAG input file becomes

```
# File name: diamond.dag
#
JOB  A  A.condor
JOB  B  B.condor
JOB  C  C.condor
JOB  D  D.condor
SCRIPT PRE  B  pre.csh $JOB .gz
SCRIPT PRE  C  pre.csh $JOB .gz
PARENT A CHILD B C
PARENT B C CHILD D
```

The script `pre.csh` uses the arguments to form the file name of the compressed file:

```
#!/bin/csh
gunzip $argv[1]$argv[2]
```

- **PARENT** . . .**CHILD**

  The *PARENT* and *CHILD* key words specify the dependencies within the DAG. Nodes are parents and/or children within the DAG. A parent node must be completed successfully before any of its children may be started. A child node may only be started once all its parents have successfully completed.

  The syntax of a dependency line within the DAG input file:

  PARENT *ParentJobName*. . . **CHILD** *ChildJobName*. . .

  The *PARENT* key word is followed by one or more *ParentJobName*s. The *CHILD* key word is followed by one or more *ChildJobName*s. Each child job depends on every parent job within the line. A single line in the input file can specify the dependencies from one or more parents to one or more children. As an example, the line

  ```
  PARENT p1 p2 CHILD c1 c2
  ```

  produces four dependencies:

  1. `p1` to `c1`
  2. `p1` to `c2`
  3. `p2` to `c1`
  4. `p2` to `c2`

2.10.3 Submit Description File Contents and Usage of Log Files

Each node in a DAG may use a unique submit description file. One key limitation is that each Condor submit description file must submit jobs described by a single cluster number. At the present time DAGMan cannot deal with a submit file producing multiple job clusters.

DAGMan enforces the dependencies within a DAG using the events recorded in the log file(s) produced by job submission to Condor. At one time, DAGMan required that all jobs within all nodes specify the same, single log file. This is no longer the case. However, if the DAG utilizes a large number of separate log files, performance may suffer. Therefore, it is better to have fewer, or even only a single log file. Unfortunately, each Stork job currently requires a separate log file.

As of Condor version 7.3.2, DAGMan's handling of log files has significantly changed to improve resource usage and efficiency. Prior to version 7.3.2, DAGMan assembled a list of all relevant log files at start up, by looking at all of the submit description files for all of the nodes. It kept the log files open for the duration of the DAG. Beginning with Condor version 7.3.2, DAGMan delays opening and using the submit description file until just before it is going to submit the job. At that point, DAGMan reads the submit description file to discover the job's log file. And, DAGMan monitors only the log files that are relevant to the jobs currently queued, or associated with nodes for which a POST script is running.

The advantages of the new "lazy log file evaluation" scheme are:

- The *condor_dagman* executable uses fewer file descriptors.
- It is much easier to have one node of a DAG produce the submit description file for a descendant node in the DAG.

There is one known disadvantage of the lazy log file evaluation scheme:

- Because the log files are internally identified by inode numbers, it is possible that errors may arise where log files for a given DAG are spread across more than one device. This permits two unique files to have the same inode number. We hope to have this problem fixed soon.

Another new feature in version 7.3.2 is the use of default node job user logs. Previously, it was a fatal error if the submit description file for a node job did not specify a log file. Starting with Condor version 7.3.2, DAGMan specifies a default user log file for any job that does not specify a log file. The file used as the default node log is controlled by the `DAGMAN_DEFAULT_NODE_LOG` configuration variable. A complete description is at section 3.3.26. Nodes specifying a log file and other nodes using the default log file can be mixed in a single DAG.

An additional restriction applies to the submit description file command **Log** specific to a Condor job within a DAG node. This command may not be defined in such a way that it uses macros. Using a macro would violate the restriction that there be exactly one log file specified for the potentially multiple jobs within a single cluster.

Here is a modified version of the DAG input file for the diamond-shaped DAG. The modification has each node use the same submit description file.

```
# File name: diamond.dag
#
JOB  A  diamond_job.condor
JOB  B  diamond_job.condor
JOB  C  diamond_job.condor
JOB  D  diamond_job.condor
PARENT A CHILD B C
PARENT B C CHILD D
```

Here is the single Condor submit description file for this DAG:

```
# File name: diamond_job.condor
#
executable   = /path/diamond.exe
output       = diamond.out.$(cluster)
error        = diamond.err.$(cluster)
log          = diamond_condor.log
universe     = vanilla
notification = NEVER
queue
```

This example uses the same Condor submit description file for all the jobs in the DAG. This implies that each node within the DAG runs the same job. The `$(cluster)` macro produces unique file names for each job's output. As the Condor job within each node causes a separate job submission, each has a unique cluster number.

Notification is set to `NEVER` in this example. This tells Condor not to send e-mail about the completion of a job submitted to Condor. For DAGs with many nodes, this reduces or eliminates excessive numbers of e-mails.

The job ClassAd attribute `DAGParentNodeNames` is also available for use within the submit description file. It defines a comma separated list of each *JobName* which is a parent node of this job's node. This attribute may be used in the **arguments** command for all but scheduler universe jobs. For example, if the job has two parents, with *JobName*s B and C, the submit description file command

```
arguments = $$([DAGParentNodeNames])
```

will pass the string "B,C" as the command line argument when invoking the job.

2.10.4 DAG Submission

A DAG is submitted using the program *condor_submit_dag*. See the manual page 859 for complete details. A simple submission has the syntax

condor_submit_dag DAGInputFileName

The diamond-shaped DAG example may be submitted with

```
condor_submit_dag diamond.dag
```

In order to guarantee recoverability, the DAGMan program itself is run as a Condor job. As such, it needs a submit description file. *condor_submit_dag* produces this needed submit description file, naming it by appending `.condor.sub` to the *DAGInputFileName*. This submit description file may be edited if the DAG is submitted with

```
condor_submit_dag -no_submit diamond.dag
```

causing *condor_submit_dag* to generate the submit description file, but not submit DAGMan to Condor. To submit the DAG, once the submit description file is edited, use

```
condor_submit diamond.dag.condor.sub
```

An optional argument to *condor_submit_dag*, *-maxjobs*, is used to specify the maximum number of batch jobs that DAGMan may submit at one time. It is commonly used when there is a limited

amount of input file staging capacity. As a specific example, consider a case where each job will require 4 Mbytes of input files, and the jobs will run in a directory with a volume of 100 Mbytes of free space. Using the argument *-maxjobs 25* guarantees that a maximum of 25 jobs, using a maximum of 100 Mbytes of space, will be submitted to Condor and/or Stork at one time.

While the *-maxjobs* argument is used to limit the number of batch system jobs submitted at one time, it may be desirable to limit the number of scripts running at one time. The optional *-maxpre* argument limits the number of PRE scripts that may be running at one time, while the optional *-maxpost* argument limits the number of POST scripts that may be running at one time.

An optional argument to *condor_submit_dag*, *-maxidle*, is used to limit the number of idle jobs within a given DAG. When the number of idle node jobs in the DAG reaches the specified value, *condor_dagman* will stop submitting jobs, even if there are ready nodes in the DAG. Once some of the idle jobs start to run, *condor_dagman* will resume submitting jobs. Note that this parameter only limits the number of idle jobs submitted by a given instance of *condor_dagman*. Idle jobs submitted by other sources (including other *condor_dagman* runs) are ignored.

2.10.5 Job Monitoring, Job Failure, and Job Removal

After submission, the progress of the DAG can be monitored by looking at the log file(s), observing the e-mail that job submission to Condor causes, or by using *condor_q -dag*. There is a large amount of information in an extra file. The name of this extra file is produced by appending `.dagman.out` to *DAGInputFileName*; for example, if the DAG file is `diamond.dag`, this extra file is `diamond.dag.dagman.out`. If this extra file grows too large, limit its size with the `MAX_DAGMAN_LOG` configuration macro (see section 3.3.4).

If you have some kind of problem in your DAGMan run, please save the corresponding `dagman.out` file; it is the most important debugging tool for DAGMan. As of version 6.8.2, the `dagman.out` is appended to, rather than overwritten, with each new DAGMan run.

condor_submit_dag attempts to check the DAG input file. If a problem is detected, *condor_submit_dag* prints out an error message and aborts.

To remove an entire DAG, consisting of DAGMan plus any jobs submitted to Condor or Stork, remove the DAGMan job running under Condor. *condor_q* will list the job number. Use the job number to remove the job, for example

```
% condor_q
-- Submitter: turunmaa.cs.wisc.edu : <128.105.175.125:36165> : turunmaa.cs.wisc.edu
 ID      OWNER          SUBMITTED     RUN_TIME ST PRI SIZE CMD
  9.0   smoler         10/12 11:47   0+00:01:32 R  0   8.7  condor_dagman -f -
 11.0   smoler         10/12 11:48   0+00:00:00 I  0   3.6  B.out
 12.0   smoler         10/12 11:48   0+00:00:00 I  0   3.6  C.out

    3 jobs; 2 idle, 1 running, 0 held

% condor_rm 9.0
```

Before the DAGMan job stops running, it uses *condor_rm* to remove any jobs within the DAG that are running.

In the case where a machine is scheduled to go down, DAGMan will clean up memory and exit. However, it will leave any submitted jobs in Condor's queue.

2.10.6 Advanced Features of DAGMan

Retrying Failed Nodes or Stopping the Entire DAG

The *RETRY* key word provides a way to retry failed nodes. The use of retry is optional. The syntax for retry is

RETRY *JobName NumberOfRetries* [**UNLESS-EXIT** *value*]

where *JobName* identifies the node. *NumberOfRetries* is an integer number of times to retry the node after failure. The implied number of retries for any node is 0, the same as not having a retry line in the file. Retry is implemented on nodes, not parts of a node.

The diamond-shaped DAG example may be modified to retry node C:

```
# File name: diamond.dag
#
JOB  A  A.condor
JOB  B  B.condor
JOB  C  C.condor
JOB  D  D.condor
PARENT A CHILD B C
PARENT B C CHILD D
Retry  C 3
```

If node C is marked as failed (for any reason), then it is started over as a first retry. The node will be tried a second and third time, if it continues to fail. If the node is marked as successful, then further retries do not occur.

Retry of a node may be short circuited using the optional key word *UNLESS-EXIT* (followed by an integer exit value). If the node exits with the specified integer exit value, then no further processing will be done on the node.

The variable `$RETRY` evaluates to an integer value set to 0 first time a node is run, and is incremented each time for each time the node is retried. The variable `$MAX_RETRIES` is the value set for *NumberOfRetries*.

The *ABORT-DAG-ON* key word provides a way to abort the entire DAG if a given node returns a specific exit code. The syntax for *ABORT-DAG-ON* is

ABORT-DAG-ON *JobName AbortExitValue* [**RETURN** *DAGReturnValue*]

If the node specified by *JobName* returns the specified *AbortExitValue*, the DAG is immediately

aborted. A DAG abort differs from a node failure, in that a DAG abort causes all nodes within the DAG to be stopped immediately. This includes removing the jobs in nodes that are currently running. A node failure allows the DAG to continue running, until no more progress can be made due to dependencies.

An abort overrides node retries. If a node returns the abort exit value, the DAG is aborted, even if the node has retry specified.

When a DAG aborts, by default it exits with the node return value that caused the abort. This can be changed by using the optional *RETURN* key word along with specifying the desired *DAGReturnValue.* The DAG abort return value can be used for DAGs within DAGs, allowing an inner DAG to cause an abort of an outer DAG.

Adding *ABORT-DAG-ON* for node C in the diamond-shaped DAG

```
# File name: diamond.dag
#
JOB  A  A.condor
JOB  B  B.condor
JOB  C  C.condor
JOB  D  D.condor
PARENT A CHILD B C
PARENT B C CHILD D
Retry  C 3
ABORT-DAG-ON C 10 RETURN 1
```

causes the DAG to be aborted, if node C exits with a return value of 10. Any other currently running nodes (only node B is a possibility for this particular example) are stopped and removed. If this abort occurs, the return value for the DAG is 1.

Variable Values Associated with Nodes

The *VARS* key word provides a method for defining a macro that can be referenced in the node's submit description file. These macros are defined on a per-node basis, using the following syntax:

VARS *JobName macroname="string"* [*macroname="string"...]*

The macro may be used within the submit description file of the relevant node. A *macroname* consists of alphanumeric characters (a..Z and 0..9), as well as the underscore character. The space character delimits macros, when there is more than one macro defined for a node.

Correct syntax requires that the *string* must be enclosed in double quotes. To use a double quote inside *string*, escape it with the backslash character (\). To add the backslash character itself, use two backslashes (\\). The string $(JOB) maybe used in *string* and will expand to *JobName*. If the *VARS* line appears in a DAG file used as a splice file, then $(JOB) will be the fully scoped name of the node.

Note that the *macroname* itself cannot begin with the string `queue`, in any combination of upper or lower case.

If the DAG input file contains

```
# File name: diamond.dag
#
JOB  A  A.condor
JOB  B  B.condor
JOB  C  C.condor
JOB  D  D.condor
VARS A state="Wisconsin"
PARENT A CHILD B C
PARENT B C CHILD D
```

then file `A.condor` may use the macro `state`. This example submit description file for the Condor job in node A passes the value of the macro as a command-line argument to the job.

```
# file name: A.condor
executable = A.exe
log        = A.log
error      = A.err
arguments  = "$(state)"
queue
```

This Condor job's command line will be

```
A.exe Wisconsin
```

The use of macros may allow a reduction in the necessary number of unique submit description files.

A separate example shows an intended use of a *VARS* entry in the DAG input file. This use may dramatically reduce the number of Condor submit description files needed for a DAG. In the case where the submit description file for each node varies only in file naming, the use of a substitution macro within the submit description file reduces the need to a single submit description file. Note that the user log file for a job currently cannot be specified using a macro passed from the DAG.

The example uses a single submit description file in the DAG input file, and uses the *VARS* entry to name output files.

The relevant portion of the DAG input file appears as

```
JOB A theonefile.sub
JOB B theonefile.sub
JOB C theonefile.sub

VARS A outfilename="A"
VARS B outfilename="B"
VARS C outfilename="C"
```

The submit description file appears as

```
# submit description file called:  theonefile.sub
executable   = progX
universe     = standard
output       = $(outfilename)
error        = error.$(outfilename)
log          = progX.log
queue
```

For a DAG such as this one, but with thousands of nodes, being able to write and maintain a single submit description file and a single, yet more complex, DAG input file is preferable.

Special characters within VARS string definitions

The value of a *VARS macroname* may contain spaces and tabs. It is also possible to have double quote marks and backslashes within these values. **Unfortunately, it is not possible to have single quote marks within these values.** In order to have spaces or tabs within a value, use the new syntax format for the **arguments** command in the node's Condor job submit description file, as described in section 9. Double quote marks are escaped differently, depending on the new syntax or old syntax argument format. Note that in both syntaxes, double quote marks require two levels of escaping: one level is for the parsing of the DAG input file, and the other level is for passing the resulting value through *condor_submit*.

As an example, here are only the relevant parts of a DAG input file. Note that the NodeA value for `second` contains a tab.

```
Vars NodeA first="Alberto Contador"
Vars NodeA second="\"\"Andy Schleck\"\""
Vars NodeA third="Lance\\ Armstrong"
Vars NodeA misc="!@#$%^&*()_-=+=[]{}?/"

Vars NodeB first="Lance_Armstrong"
Vars NodeB second="\\\"Andreas_Kloden\\\""
Vars NodeB third="Ivan\\_Basso"
Vars NodeB misc="!@#$%^&*()_-=+=[]{}?/"
```

The new syntax **arguments** line of the Condor submit description file for NodeA is

```
arguments = "'$(first)' '$(second)' '$(third)' '$(misc)'"
```

The single quotes around each variable reference are only necessary if the variable value may contain spaces or tabs. The resulting values passed to the NodeA executable are

```
Alberto Contador
"Andy Schleck"
Lance\ Armstrong
!@#$%^&*()_-=+=[]{}?/
```

The old syntax **arguments** line of the Condor submit description file for NodeB is

```
arguments = $(first) $(second) $(third) $(misc)
```

The resulting values passed to the NodeB executable are

```
Lance_Armstrong
"Andreas_Kloden"
Ivan\_Basso
!@#$%^&*()_-=+=[]{}?/
```

Setting Priorities for Nodes

The *PRIORITY* key word assigns a priority to a DAG node. The syntax for *PRIORITY* is

PRIORITY *JobName PriorityValue*

The node priority affects the order in which nodes that are ready at the same time will be submitted. Note that node priority does *not* override the DAG dependencies.

Node priority is mainly relevant if node submission is throttled via the *-maxjobs* or *-maxidle* command-line arguments or the `DAGMAN_MAX_JOBS_SUBMITTED` or `DAGMAN_MAX_JOBS_IDLE` configuration variables. Note that PRE scripts can affect the order in which jobs run, so DAGs containing PRE scripts may not run the nodes in exact priority order, even if doing so would satisfy the DAG dependencies.

The priority value is an integer (which can be negative). A larger numerical priority is better (will be run before a smaller numerical value). The default priority is 0.

Adding *PRIORITY* for node C in the diamond-shaped DAG

```
# File name: diamond.dag
#
JOB  A  A.condor
JOB  B  B.condor
JOB  C  C.condor
JOB  D  D.condor
PARENT A CHILD B C
PARENT B C CHILD D
Retry  C 3
PRIORITY C 1
```

This will cause node C to be submitted before node B (normally, node B would be submitted first).

Limiting the Number of Submitted Job Clusters within a DAG

In order to limit the number of submitted job clusters within a DAG, the nodes may be placed into categories by assignment of a name. Then, a maximum number of submitted clusters may be specified for each category.

The *CATEGORY* key word assigns a category name to a DAG node. The syntax for *CATEGORY* is

CATEGORY *JobName CategoryName*

Category names cannot contain white space.

The *MAXJOBS* key word limits the number of submitted job clusters on a per category basis. The syntax for *MAXJOBS* is

MAXJOBS *CategoryName MaxJobsValue*

If the number of submitted job clusters for a given category reaches the limit, no further job clusters in that category will be submitted until other job clusters within the category terminate. If MAXJOBS is not set for a defined category, then there is no limit placed on the number of submissions within that category.

Note that a single invocation of *condor_submit* results in one job cluster. The number of Condor jobs within a cluster may be greater than 1.

The configuration variable `DAGMAN_MAX_JOBS_SUBMITTED` and the *condor_submit_dag -maxjobs* command-line option are still enforced if these *CATEGORY* and *MAXJOBS* throttles are used.

Please see the end of section 2.10.6 on DAG Splicing for a description of the interaction between categories and splices.

Configuration Specific to a DAG

The *CONFIG* keyword specifies a configuration file to be used to set configuration variables related to *condor_dagman* when running this DAG. The syntax for *CONFIG* is

CONFIG *ConfigFileName*

If the DAG file contains a line like this,

```
CONFIG dagman.config
```

then the configuration values in the file `dagman.config` will be used for this DAG.

Configuration macros for *condor_dagman* can be specified in several ways, as given within the ordered list:

1. In a Condor configuration file.
2. With an environment variable. Prepend the string `_CONDOR_` to the configuration variable's name.
3. As specified above, with a line in the DAG input file using the keyword *CONFIG*, such that there is a *condor_dagman*-specific configuration file specified, or on the *condor_submit_dag* command line.

4. For some configuration variables, there is a corresponding *condor_submit_dag* command line argument. For example, the configuration variable `DAGMAN_MAX_JOBS_SUBMITTED` has the corresponding command line argument *-maxjobs*.

In the above list, configuration values specified later in the list override ones specified earlier For example, a value specified on the *condor_submit_dag* command line overrides corresponding values in any configuration file. And, a value specified in a DAGMan-specific configuration file overrides values specified in a general Condor configuration file.

Configuration variables that are not for *condor_dagman* and not utilized by DaemonCore, yet are specified in a *condor_dagman*-specific configuration file are ignored.

Only a single configuration file can be specified for a given *condor_dagman* run. For example, if one file is specified within a DAG input file, and a different file is specified on the *condor_submit_dag* command line, this is a fatal error at submit time. The same is true if different configuration files are specified in multiple DAG input files, and referenced in a single *condor_submit_dag* command.

If multiple DAGs are run in a single *condor_dagman* run, the configuration options specified in the *condor_dagman* configuration file, if any, apply to all DAGs, even if some of the DAGs specify no configuration file.

Configuration variables relating to DAGMan may be found in section 3.3.26.

Single Submission of Multiple, Independent DAGs

A single use of *condor_submit_dag* may execute multiple, independent DAGs. Each independent DAG has its own DAG input file. These DAG input files are command-line arguments to *condor_submit_dag* (see the *condor_submit_dag* manual page at 9).

Internally, all of the independent DAGs are combined into a single, larger DAG, with no dependencies between the original independent DAGs. As a result, any generated rescue DAG file represents all of the input DAGs as a single DAG. The file name of this rescue DAG is based on the DAG input file listed first within the command-line arguments to *condor_submit_dag* (unlike a single-DAG rescue DAG file, however, the file name will be `<whatever>.dag_multi.rescue` or `<whatever>.dag_multi.rescueNNN`, as opposed to just `<whatever>.dag.rescue` or `<whatever>.dag.rescueNNN`). Other files such as `dagman.out` and the lock file also have names based on this first DAG input file.

The success or failure of the independent DAGs is well defined. When multiple, independent DAGs are submitted with a single command, the success of the composite DAG is defined as the logical AND of the success of each independent DAG. This implies that failure is defined as the logical OR of the failure of any of the independent DAGs.

By default, DAGMan internally renames the nodes to avoid node name collisions. If all node names are unique, the renaming of nodes may be disabled by setting the configuration variable `DAGMAN_MUNGE_NODE_NAMES` to `False` (see 3.3.26).

A DAG Within a DAG Is a SUBDAG

The organization and dependencies of the jobs within a DAG are the keys to its utility. Some DAGs are naturally constructed hierarchically, such that a node within a DAG is also a DAG. Condor DAGMan handles this situation easily. DAGs can be nested to any depth.

One of the highlights of using the SUBDAG feature is that portions of a DAG may be constructed and modified during the execution of the DAG. A drawback may be that each SUBDAG causes its own distinct job submission of *condor_dagman*, leading to a larger number of jobs, together with their potential need of carefully constructed policy configuration to throttle node submission or execution.

Since more than one DAG is being discussed, here is terminology introduced to clarify which DAG is which. Reuse the example diamond-shaped DAG as given in Figure 2.3. Assume that node B of this diamond-shaped DAG will itself be a DAG. The DAG of node B is called a SUBDAG, inner DAG, or lower-level DAG. The diamond-shaped DAG is called the outer or top-level DAG.

Work on the inner DAG first. Here is a very simple linear DAG input file used as an example of the inner DAG.

```
# File name: inner.dag
#
JOB  X  X.submit
JOB  Y  Y.submit
JOB  Z  Z.submit
PARENT X CHILD Y
PARENT Y CHILD Z
```

The Condor submit description file, used by *condor_dagman*, corresponding to `inner.dag` will be named `inner.dag.condor.sub`. The DAGMan submit description file is always named `<DAG file name>.condor.sub`. Each DAG or SUBDAG results in the submission of *condor_dagman* as a Condor job, and *condor_submit_dag* creates this submit description file.

The preferred presentation of the DAG input file for the outer DAG is

```
# File name: diamond.dag
#
    JOB  A  A.submit
    SUBDAG EXTERNAL  B  inner.dag
    JOB  C  C.submit
    JOB  D  D.submit
    PARENT A CHILD B C
    PARENT B C CHILD D
```

The preferred presentation is equivalent to

```
# File name: diamond.dag
```

```
#
    JOB  A  A.submit
    JOB  B  inner.dag.condor.sub
    JOB  C  C.submit
    JOB  D  D.submit
    PARENT A CHILD B C
    PARENT B C CHILD D
```

Within the outer DAG's input file, the **SUBDAG** keyword specifies a special case of a **JOB** node, where the job is itself a DAG.

The syntax for each SUBDAG entry is

SUBDAG EXTERNAL *JobName DagFileName* [**DIR** *directory*] [**NOOP**] [**DONE**]

The optional specifications of **DIR**, **NOOP**, and **DONE**, if used, must appear in this order within the entry.

A **SUBDAG** node is essentially the same as any other node, except that the DAG input file for the inner DAG is specified, instead of the Condor submit file. The keyword **EXTERNAL** means that the SUBDAG is run within its own instance of *condor_dagman*.

NOOP and **DONE** for **SUBDAG** nodes have the same effect that they do for **JOB** nodes.

Here are details that affect SUBDAGs:

- Nested Submit Description File Generation

  There are three ways to generate the `<DAG file name>.condor.sub` file of a SUBDAG:

  - **Lazily** (the default in Condor version 7.5.2 and later versions)
  - **Eagerly** (the default in Condor versions 7.4.1 through 7.5.1)
  - **Manually** (the only way prior to version Condor version 7.4.1)

  When the `<DAG file name>.condor.sub` file is generated **lazily**, this file is generated immediately before the SUBDAG job is submitted. Generation is accomplished by running

  ```
  condor_submit_dag -no_submit
  ```

  on the DAG input file specified in the **SUBDAG** entry. This is the default behavior. There are advantages to this lazy mode of submit description file creation for the SUBDAG:

  - The DAG input file for a SUBDAG does not have to exist until the SUBDAG is ready to run, so this file can be dynamically created by earlier parts of the outer DAG or by the PRE script of the node containing the SUBDAG.
  - It is now possible to have SUBDAGs within splices. That is not possible with eager submit description file creation, because *condor_submit_dag* does not understand splices.

The main disadvantage of lazy submit file generation is that a syntax error in the DAG input file of a SUBDAG will not be discovered until the outer DAG tries to run the inner DAG.

When `<DAG file name>.condor.sub` files are generated **eagerly**, *condor_submit_dag* runs itself recursively (with the *-no_submit* option) on each SUBDAG, so all of the `<DAG file name>.condor.sub` files are generated before the top-level DAG is actually submitted. To generate the `<DAG file name>.condor.sub` files eagerly, pass the *-do_recurse* flag to *condor_submit_dag*; also set the `DAGMAN_GENERATE_SUBDAG_SUBMITS` configuration variable to `False`, so that *condor_dagman* does not re-run *condor_submit_dag* at run time thereby regenerating the submit description files.

To generate the `.condor.sub` files **manually**, run

```
condor_submit_dag -no_submit
```

on each lower-level DAG file, before running *condor_submit_dag* on the top-level DAG file; also set the `DAGMAN_GENERATE_SUBDAG_SUBMITS` configuration variable to `False`, so that *condor_dagman* does not re-run *condor_submit_dag* at run time. The main reason for generating the `<DAG file name>.condor.sub` files manually is to set options for the lower-level DAG that one would not otherwise be able to set An example of this is the *-insert_sub_file* option. For instance, using the given example do the following to manually generate Condor submit description files:

```
condor_submit_dag -no_submit -insert_sub_file fragment.sub inner.dag
condor_submit_dag diamond.dag
```

Note that most *condor_submit_dag* command-line flags have corresponding configuration variables, so we encourage the use of per-DAG configuration files, especially in the case of nested DAGs. This is the easiest way to set different options for different DAGs in an overall workflow.

It is possible to combine more than one method of generating the `<DAG file name>.condor.sub` files. For example, one might pass the *-do_recurse* flag to *condor_submit_dag*, but leave the `DAGMAN_GENERATE_SUBDAG_SUBMITS` configuration variable set to the default of `True`. Doing this would provide the benefit of an immediate error message at submit time, if there is a syntax error in one of the inner DAG input files, but the lower-level `<DAG file name>.condor.sub` files would still be regenerated before each nested DAG is submitted.

The values of the following command-line flags are passed from the top-level *condor_submit_dag* instance to any lower-level *condor_submit_dag* instances. This occurs whether the lower-level submit description files are generated lazily or eagerly:

- **-verbose**
- **-force**
- **-notification**
- **-allowlogerror**

- **-dagman**
- **-usedagdir**
- **-outfile_dir**
- **-oldrescue**
- **-autorescue**
- **-dorescuefrom**
- **-allowversionmismatch**
- **-no_recurse/do_recurse**
- **-update_submit**
- **-import_env**

The values of the following command-line flags are preserved in any already-existing lower-level DAG submit description files:

- **-maxjobs**
- **-maxidle**
- **-maxpre**
- **-maxpost**
- **-debug**

Other command-line arguments are set to their defaults in any lower-level invocations of *condor_submit_dag*.

The **-force** option will cause existing DAG submit description files to be overwritten without preserving any existing values.

- Submission of the outer DAG

  The outer DAG is submitted as before, with the command

  ```
  condor_submit_dag diamond.dag
  ```

- Interaction with Rescue DAGs

  When using nested DAGs, we strongly recommend that you use "new-style" rescue DAGs. This is the default. Using "new-style" rescue DAGs will automatically run the proper rescue DAG(s) if there is a failure in the work flow. For example, if one of the nodes in `inner.dag` fails, this will produce a rescue DAG for inner.dag (named `inner.dag.rescue.001`, etc.). Then, since `inner.dag` failed, node B of `diamond.dag` will fail, producing a rescue DAG for `diamond.dag` (named `diamond.dag.rescue.001`, etc.). If the command

  ```
  condor_submit_dag diamond.dag
  ```

  is re-run, the most recent outer rescue DAG will be run, and this will re-run the inner DAG, which will in turn run the most recent inner rescue DAG. The use of "old-style" rescue DAGs will require the renaming of the inner rescue DAG or manually running it.

- File Paths

  Remember that, unless the DIR keyword is used in the outer DAG, the inner DAG utilizes the current working directory when the outer DAG is submitted. Therefore, all paths utilized by the inner DAG file must be specified accordingly.

DAG Splicing

A weakness in scalability exists when submitting a DAG within a DAG. Each executing independent DAG requires its own invocation of *condor_dagman* to be running. The scaling issue presents itself when the same semantic DAG is reused hundreds or thousands of times in a larger DAG. Further, there may be many rescue DAGs created if a problem occurs. To alleviate these concerns, the DAGMan language introduces the concept of graph splicing.

A splice is a named instance of a subgraph which is specified in a separate DAG file. The splice is treated as a whole entity during dependency specification in the including DAG. The same DAG file may be reused as differently named splices, each one incorporating a copy of the dependency graph (and nodes therein) into the including DAG. Any splice in an including DAG may have dependencies between the sets of initial and final nodes. A splice may be incorporated into an including DAG without any dependencies; it is considered a disjoint DAG within the including DAG. The nodes within a splice are scoped according to a hierarchy of names associated with the splices, as the splices are parsed from the top level DAG file. The scoping character to describe the inclusion hierarchy of nodes into the top level dag is `'+'`. This character is chosen due to a restriction in the allowable characters which may be in a file name across the variety of ports that Condor supports. In any DAG file, all splices must have unique names, but the same splice name may be reused in different DAG files.

Condor does not detect nor support splices that form a cycle within the DAG. A DAGMan job that causes a cyclic inclusion of splices will eventually exhaust available memory and crash.

The *SPLICE* keyword in a DAG input file creates a named instance of a DAG as specified in another file as an entity which may have *PARENT* and *CHILD* dependencies associated with other splice names or node names in the including DAG file. The syntax for *SPLICE* is

SPLICE *SpliceName DagFileName* [**DIR** *directory*]

After parsing incorporates a splice, all nodes within the spice become nodes within the including DAG.

The following series of examples illustrate potential uses of splicing. To simplify the examples, presume that each and every job uses the same, simple Condor submit description file:

```
# BEGIN SUBMIT FILE submit.condor
executable   = /bin/echo
arguments    = OK
universe     = vanilla
output       = $(jobname).out
```

```
error        = $(jobname).err
log          = submit.log
notification = NEVER
queue
# END SUBMIT FILE submit.condor
```

This first simple example splices a diamond-shaped DAG in between the two nodes of a top level DAG. Here is the DAG input file for the diamond-shaped DAG:

```
# BEGIN DAG FILE diamond.dag
JOB A submit.condor
VARS A jobname="$(JOB)"

JOB B submit.condor
VARS B jobname="$(JOB)"

JOB C submit.condor
VARS C jobname="$(JOB)"

JOB D submit.condor
VARS D jobname="$(JOB)"

PARENT A CHILD B C
PARENT B C CHILD D
# END DAG FILE diamond.dag
```

The top level DAG incorporates the diamond-shaped splice:

```
# BEGIN DAG FILE toplevel.dag
JOB X submit.condor
VARS X jobname="$(JOB)"

JOB Y submit.condor
VARS Y jobname="$(JOB)"

# This is an instance of diamond.dag, given the symbolic name DIAMOND
SPLICE DIAMOND diamond.dag

# Set up a relationship between the nodes in this dag and the splice

PARENT X CHILD DIAMOND
PARENT DIAMOND CHILD Y

# END DAG FILE toplevel.dag
```

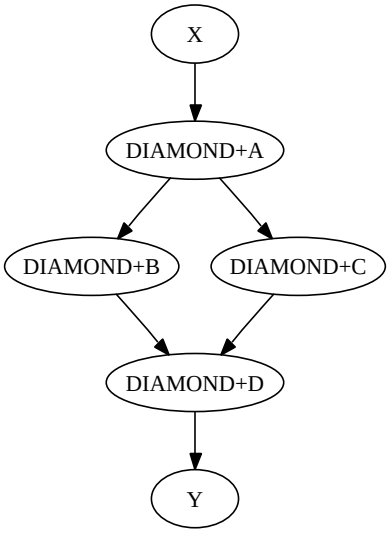

Figure 2.4: The diamond-shaped DAG spliced between two nodes.

Figure 2.4 illustrates the resulting top level DAG and the dependencies produced. Notice the naming of nodes scoped with the splice name. This hierarchy of splice names assures unique names associated with all nodes.

Figure 2.5 illustrates the starting point for a more complex example. The DAG input file `X.dag` describes this X-shaped DAG. The completed example displays more of the spatial constructs provided by splices. Pay particular attention to the notion that each named splice creates a new graph, even when the same DAG input file is specified.

```
# BEGIN DAG FILE X.dag

JOB A submit.condor
VARS A jobname="$(JOB)"

JOB B submit.condor
VARS B jobname="$(JOB)"

JOB C submit.condor
VARS C jobname="$(JOB)"

JOB D submit.condor
VARS D jobname="$(JOB)"

JOB E submit.condor
VARS E jobname="$(JOB)"
```

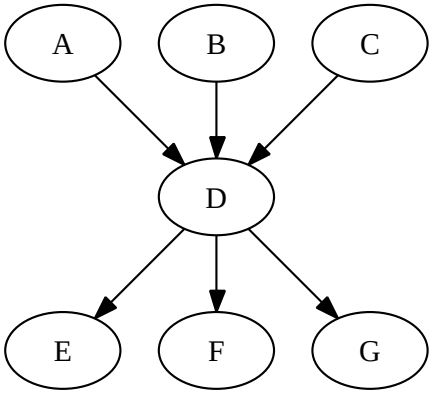

Figure 2.5: The X-shaped DAG.

```
JOB F submit.condor
VARS F jobname="$(JOB)"

JOB G submit.condor
VARS G jobname="$(JOB)"

# Make an X-shaped dependency graph
PARENT A B C CHILD D
PARENT D CHILD E F G

# END DAG FILE X.dag
```

File `s1.dag` continues the example, presenting the DAG input file that incorporates two separate splices of the X-shaped DAG. Figure 2.6 illustrates the resulting DAG.

```
# BEGIN DAG FILE s1.dag

JOB A submit.condor
VARS A jobname="$(JOB)"

JOB B submit.condor
VARS B jobname="$(JOB)"

# name two individual splices of the X-shaped DAG
SPLICE X1 X.dag
SPLICE X2 X.dag

# Define dependencies
# A must complete before the initial nodes in X1 can start
PARENT A CHILD X1
```

```
# All final nodes in X1 must finish before
# the initial nodes in X2 can begin
PARENT X1 CHILD X2
# All final nodes in X2 must finish before B may begin.
PARENT X2 CHILD B

# END DAG FILE s1.dag
```

The top level DAG in the hierarchy of this complex example is described by the DAG input file `toplevel.dag`. Figure 2.7 illustrates the final DAG. Notice that the DAG has two disjoint graphs in it as a result of splice S3 not having any dependencies associated with it in this top level DAG.

```
# BEGIN DAG FILE toplevel.dag

JOB A submit.condor
VARS A jobname="$(JOB)"

JOB B submit.condor
VARS B jobname="$(JOB)"

JOB C submit.condor
VARS C jobname="$(JOB)"

JOB D submit.condor
VARS D jobname="$(JOB)"

# a diamond-shaped DAG
PARENT A CHILD B C
PARENT B C CHILD D

# This splice of the X-shaped DAG can only run after
# the diamond dag finishes
SPLICE S2 X.dag
PARENT D CHILD S2

# Since there are no dependencies for S3,
# the following splice is disjoint
SPLICE S3 s1.dag

# END DAG FILE toplevel.dag
```

The *DIR* option specifies a working directory for a splice, from which the splice will be parsed and the containing jobs submitted. The directory associated with the splices' *DIR* specification will be propagated as a prefix to all nodes in the splice and any included splices. If a node already has a

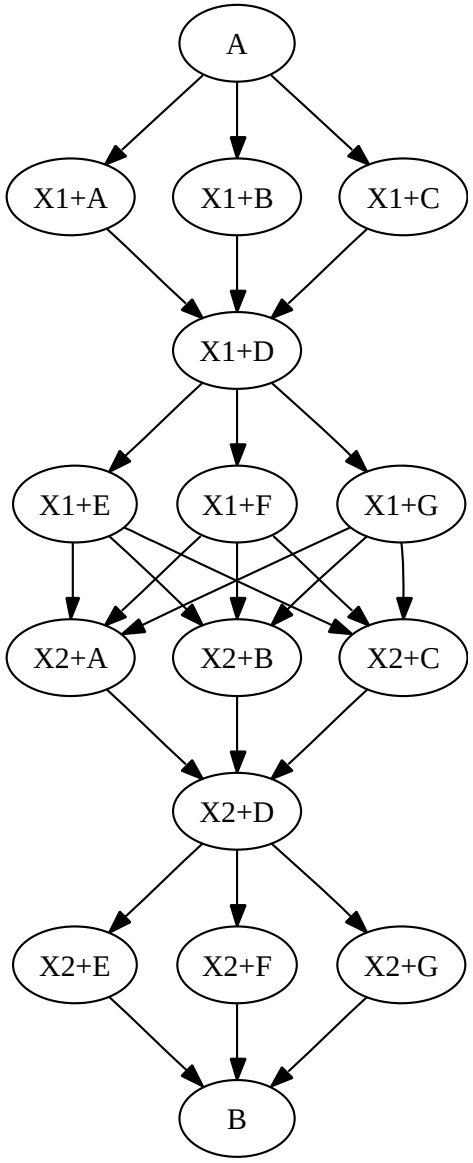

Figure 2.6: The DAG described by `s1.dag`.

DIR specification, then the splice's *DIR* specification will be a prefix to the nodes and separated by a directory separator character. Jobs in included splices with an absolute path for their *DIR* specification will have their *DIR* specification untouched. Note that a DAG containing *DIR* specifications cannot be run in conjunction with the *-usedagdir* command-line argument to *condor_submit_dag*. A rescue DAG generated by a DAG run with the *-usedagdir* argument will contain DIR specifications, so the rescue DAG must be run *without* the *-usedagdir* argument.

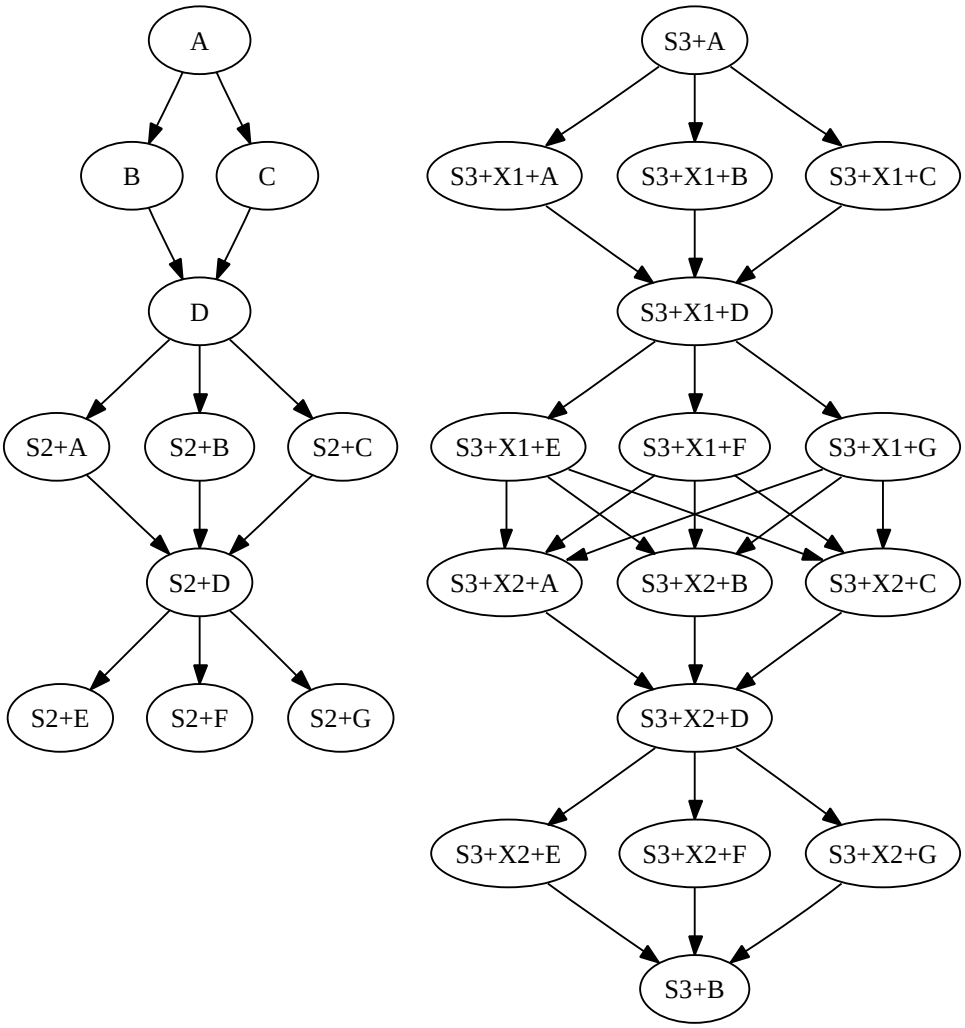

Figure 2.7: The complex splice example DAG.

The Interaction of Categories and MAXJOBS with Splices

Categories normally refer only to nodes within a given splice. All of the assignments of nodes to a category, and the setting of the category throttle, should be done within a single DAG file. However, it is now possible to have categories include nodes from within more than one splice. To do this, the category name is prefixed with the '+' (plus) character. This tells DAGMan that the category is a cross-splice category. Towards deeper understanding, what this really does is prevent renaming of the category when the splice is incorporated into the upper-level DAG. The MAXJOBS specification for the category can appear in either the upper-level DAG file or one of the splice DAG files. It probably makes the most sense to put it in the upper-level DAG file.

Here is an example which applies a single limitation on submitted jobs, identifying the category with `+init`.

```
# relevant portion of file name: upper.dag

    SPLICE A splice1.dag
    SPLICE B splice2.dag

    MAXJOBS +init 2

# relevant portion of file name: splice1.dag

    JOB C C.sub
    CATEGORY C +init
    JOB D D.sub
    CATEGORY D +init


# relevant portion of file name: splice2.dag

    JOB X X.sub
    CATEGORY X +init
    JOB Y Y.sub
    CATEGORY Y +init
```

For both global and non-global category throttles, settings at a higher level in the DAG override settings at a lower level. In this example:

```
# relevant portion of file name: upper.dag

    SPLICE A lower.dag

    MAXJOBS A+catX 10
    MAXJOBS +catY 2


# relevant portion of file name: lower.dag

    MAXJOBS catX 5
    MAXJOBS +catY 1
```

the resulting throttle settings are 2 for the `+catY` category and 10 for the `A+catX` category in

splice. Note that non-global category names are prefixed with their splice name(s), so to refer to a non-global category at a higher level, the splice name must be included.

2.10.7 Job Recovery: The Rescue DAG

DAGMan can help with the resubmission of uncompleted portions of a DAG, when one or more nodes result in failure. If any node in the DAG fails, the remainder of the DAG is continued until no more forward progress can be made based on the DAG's dependencies. At this point, DAGMan produces a file called a Rescue DAG.

The Rescue DAG is a DAG input file, functionally the same as the original DAG input file. The Rescue DAG additionally contains an indication of successfully completed nodes by appending the *DONE* key word to the node's *JOB* or *DATA* lines. If the DAG is resubmitted utilizing the Rescue DAG, the successfully completed nodes will not be re-executed.

Note that if multiple DAG input files are specified on the *condor_submit_dag* command line, a single Rescue DAG encompassing all of the input DAGs is generated.

If the Rescue DAG file is generated before all retries of a node are completed, then the Rescue DAG file will also contain *Retry* entries. The number of retries will be set to the appropriate remaining number of retries.

The granularity defining success or failure in the Rescue DAG is the node. For a node that fails, all parts of the node will be re-run, even if some parts were successful the first time. For example, if a node's PRE script succeeds, but then the node's Condor job cluster fails, the entire node, which includes the PRE script will be re-run. A job cluster may result in the submission of multiple Condor jobs. If one of the multiple jobs fails, the node fails. Therefore, the Rescue DAG will re-run the entire node, implying the submission of the entire cluster of jobs, not just the one(s) that failed.

Statistics about the failed DAG execution are presented as comments at the beginning of the Rescue DAG input file.

The Rescue DAG is automatically generated by *condor_dagman* when a node within the DAG fails or when *condor_dagman* itself is removed with *condor_rm*. The file name of the Rescue DAG, and usage of the Rescue DAG changed from explicit specification to implicit usage beginning with Condor version 7.1.0. Current naming of the Rescue DAG appends the string `.rescue<XXX>` to the original DAG input file name. Values for `<XXX>` start at `001` and continue to `002`, `003`, and beyond. If a Rescue DAG exists, the Rescue DAG with the largest magnitude value for `<XXX>` will be used, and its usage is implied.

Here is an example showing file naming and DAG submission for the case of a failed DAG. The initial DAG is submitted with

```
condor_submit_dag  my.dag
```

A failure of this DAG results in the Rescue DAG named `my.dag.rescue001`. The DAG is resubmitted using the same command:

```
condor_submit_dag  my.dag
```

This resubmission of the DAG uses the Rescue DAG file `my.dag.rescue001`, because it exists. Failure of this Rescue DAG results in another Rescue DAG called `my.dag.rescue002`. If the DAG is again submitted, using the same command as with the first two submissions, but not repeated here, then this third submission uses the Rescue DAG file `my.dag.rescue002`, because it exists, and because the value `002` is larger in magnitude than `001`.

To explicitly specify a particular Rescue DAG, use the optional command-line argument *-dorescuefrom* with *condor_submit_dag*. Note that this will have the side effect of renaming existing Rescue DAG files with larger magnitude values of `<XXX>`. Each renamed file has its existing name appended with the string `.old`. For example, assume that `my.dag` has failed 4 times, resulting in the Rescue DAGs named `my.dag.rescue001`, `my.dag.rescue002`, `my.dag.rescue003`, and `my.dag.rescue004`. A decision is made to re-run using `my.dag.rescue002`. The submit command is

```
condor_submit_dag  -dorescuefrom 2  my.dag
```

The DAG specified by the DAG input file `my.dag.rescue002` is submitted. And, the existing Rescue DAG `my.dag.rescue003` is renamed to be `my.dag.rescue003.old`, while the existing Rescue DAG `my.dag.rescue004` is renamed to be `my.dag.rescue004.old`.

The configuration variable `DAGMAN_MAX_RESCUE_NUM` sets a maximum value for `XXX`. See section 3.3.26 for the complete definition of this configuration variable.

Rescue DAG Generated when there are Parse Errors

Starting in Condor version 7.5.5, the **-DumpRescue** option to either *condor_dagman* or *condor_submit_dag* outputs a file, even if the parsing of a DAG input file fails. In this parse failure case, *condor_dagman* produces a specially named Rescue DAG containing whatever it had successfully parsed up until the point of the parse error. This Rescue DAG may be useful in debugging parse errors in complex DAGs, especially ones using splices. This incomplete Rescue DAG is not meant to be used when resubmitting a failed DAG.

To avoid confusion between this incomplete Rescue DAG generated in the case of a parse failure and a usable Rescue DAG, a different name is given to the incomplete Rescue DAG. The name appends the string `.parse_failed` to the original DAG input file name. Therefore, if the submission of a DAG with

```
condor_submit_dag  my.dag
```

has a parse failure, the resulting incomplete Rescue DAG will be named `my.dag.parse_failed`.

To further prevent one of these incomplete Rescue DAG files from being used, a line within the file contains the single keyword *REJECT*. This causes *condor_dagman* to reject the DAG, if used as

a DAG input file. This is done because the incomplete Rescue DAG may be a syntactically correct DAG input file. It will be incomplete relative to the original DAG, such that if the incomplete Rescue DAG could be run, it could erroneously be perceived as having successfully executed the desired workflow, when, in fact, it did not.

Outdated Naming of Rescue DAG

Prior to Condor version 7.1.0, the naming of a Rescue DAG appended the string `.rescue` to the existing DAG input file name. And, the Rescue DAG file would be explicitly placed in the command line that submitted it. For example, a first submission

```
condor_submit_dag  my.dag
```

Assuming that this DAG failed, the file `my.dag.rescue` would be created. To run this Rescue DAG, the submission command is

```
condor_submit_dag  my.dag.rescue
```

If this Rescue DAG also failed, a new Rescue DAG named `my.dag.rescue.rescue` would be created.

The behavior of DAGMan with respect to Rescue DAGs can be forced to this outdated behavior by setting the configuration variables `DAGMAN_OLD_RESCUE` to `True` and `DAGMAN_AUTO_RESCUE` to `False`. See 3.3.26 and 3.3.26 for complete definitions of these configuration variables.

2.10.8 File Paths in DAGs

By default, *condor_dagman* assumes that all relative paths in a DAG input file and the associated Condor submit description files are relative to the current working directory when *condor_submit_dag* is run. Note that relative paths in submit description files can be modified by the submit command **initialdir**; see the *condor_submit* manual page within Chapter 9 for more details. The rest of this discussion ignores **initialdir**.

In most cases, path names relative to the current working directory is the desired behavior. However, if running multiple DAGs with a single *condor_dagman*, and each DAG is in its own directory, this will cause problems. In this case, use the *-usedagdir* command-line argument to *condor_submit_dag* (see the *condor_submit_dag* manual page within Chapter 9 for more details). This tells *condor_dagman* to run each DAG as if *condor_submit_dag* had been run in the directory in which the relevant DAG file exists.

For example, assume that a directory called `parent` contains two subdirectories called `dag1` and `dag2`, and that `dag1` contains the DAG input file `one.dag` and `dag2` contains the DAG input file `two.dag`. Further, assume that each DAG is set up to be run from its own directory with the following command:

```
cd dag1; condor_submit_dag one.dag
```

This will correctly run `one.dag`.

The goal is to run the two, independent DAGs located within `dag1` and `dag2` while the current working directory is `parent`. To do so, run the following command:

```
condor_submit_dag -usedagdir dag1/one.dag dag2/two.dag
```

Of course, if all paths in the DAG input file(s) and the relevant submit description files are absolute, the *-usedagdir* argument is not needed; however, using absolute paths is NOT generally a good idea.

If you *do not* use *-usedagdir*, relative paths can still work for multiple DAGs, if all file paths are given relative to the current working directory as *condor_submit_dag* is executed. However, this means that, if the DAGs are in separate directories, they cannot be submitted from their own directories, only from the parent directory the paths are set up for.

Note that if you use the *-usedagdir* argument, and your run results in a rescue DAG, the rescue DAG file will be written to the current working directory, and should be run from that directory. The rescue DAG includes all the path information necessary to run each node job in the proper directory.

2.10.9 Visualizing DAGs with *dot*

It can be helpful to see a picture of a DAG. DAGMan can assist you in visualizing a DAG by creating the input files used by the AT&T Research Labs *graphviz* package. *dot* is a program within this package, available from http://www.graphviz.org/, and it is used to draw pictures of DAGs.

DAGMan produces one or more dot files as the result of an extra line in a DAGMan input file. The line appears as

```
DOT dag.dot
```

This creates a file called `dag.dot`. which contains a specification of the DAG before any jobs within the DAG are submitted to Condor. The `dag.dot` file is used to create a visualization of the DAG by using this file as input to *dot*. This example creates a Postscript file, with a visualization of the DAG:

```
dot -Tps dag.dot -o dag.ps
```

Within the DAGMan input file, the DOT command can take several optional parameters:

- **UPDATE** This will update the dot file every time a significant update happens.

- **DONT-UPDATE** Creates a single dot file, when the DAGMan begins executing. This is the default if the parameter **UPDATE** is not used.
- **OVERWRITE** Overwrites the dot file each time it is created. This is the default, unless **DONT-OVERWRITE** is specified.
- **DONT-OVERWRITE** Used to create multiple dot files, instead of overwriting the single one specified. To create file names, DAGMan uses the name of the file concatenated with a period and an integer. For example, the DAGMan input file line

  ```
  DOT dag.dot DONT-OVERWRITE
  ```

  causes files `dag.dot.0`, `dag.dot.1`, `dag.dot.2`, etc. to be created. This option is most useful when combined with the **UPDATE** option to visualize the history of the DAG after it has finished executing.
- **INCLUDE** *path-to-filename* Includes the contents of a file given by `path-to-filename` in the file produced by the **DOT** command. The include file contents are always placed after the line of the form `label=`. This may be useful if further editing of the created files would be necessary, perhaps because you are automatically visualizing the DAG as it progresses.

If conflicting parameters are used in a DOT command, the last one listed is used.

2.10.10 Capturing the Status of Nodes in a File

DAGMan can capture the status of all DAG nodes, such that the user or a script may easily monitor the status of all DAG nodes. A node status file is periodically rewritten by DAGMan. To enable this feature, the DAG input file contains a line with the *NODE_STATUS_FILE* key word.

The syntax for a *NODE_STATUS_FILE* specification is

NODE_STATUS_FILE *statusFileName* [*minimumUpdateTime*]

The status file is written on the machine where the DAG is submitted; its location is given by *statusFileName*. This will be the same machine where the *condor_dagman* job is running.

The optional *minimumUpdateTime* specifies the minimum number of seconds that must elapse between updates to the node status file. This setting exists to avoid having DAGMan spend too much time writing the node status file for very large DAGs. If no value is specified, no limit is set. The node status file can be updated at most once per `DAGMAN_USER_LOG_SCAN_INTERVAL`, as defined at section 3.3.26, no matter how small the *minimumUpdateTime* value.

As an example, if the DAG input file contains the line

```
NODE_STATUS_FILE my.dag.status 30
```

the file `my.dag.status` will be rewritten at intervals of 30 seconds or more.

This node status file is overwritten each time it is updated. Therefore, it only holds information about the *current* status of each node; it does not provide a history of the node status. The file contains one line describing the status of every node in the DAG. The file contents do not distinguish between Condor jobs and Stork jobs. Here is an example of a node status file:

```
BEGIN 1281041745 (Thu Aug  5 15:55:45 2010)
Status of nodes of DAG(s): my.dag

JOB A STATUS_DONE      ()
JOB B STATUS_SUBMITTED (not_idle)
JOB C STATUS_SUBMITTED (idle)
JOB D STATUS_UNREADY   ()

DAG status: STATUS_SUBMITTED ()
Next scheduled update: 1281041775 (Thu Aug  5 15:56:15 2010)
END 1281041745 (Thu Aug  5 15:55:45 2010)
```

Possible node status values are:

- `STATUS_UNREADY` At least one parent has not yet finished.
- `STATUS_READY` All parents have finished, but not yet running.
- `STATUS_PRERUN` The PRE script is running.
- `STATUS_SUBMITTED` The node's Condor or Stork job(s) are in the queue.
- `STATUS_POSTRUN` The POST script is running.
- `STATUS_DONE` The node has completed successfully.
- `STATUS_ERROR` The node has failed.

A *NODE_STATUS_FILE* key word inside any splice is ignored. If multiple DAG files are specified on the *condor_submit_dag* command line, and more than one specifies a node status file, the first specification takes precedence.

2.10.11 A Machine-Readable Event History, the jobstate.log File

DAGMan can produce a machine-readable history of events. The `jobstate.log` file is designed for use by the Pegasus Workflow Management System, which operates as a layer on top of DAGMan. Pegasus uses the `jobstate.log` file to monitor the state of a workflow. The `jobstate.log` file can used by any automated tool for the monitoring of workflows.

DAGMan produces this file when the keyword *JOBSTATE_LOG* is in the DAG input file. The syntax for *JOBSTATE_LOG* is

JOBSTATE_LOG *JobstateLogFileName*

No more than one `jobstate.log` file can be created by a single instance of *condor_dagman*. If more than one `jobstate.log` file is specified, the first file name specified will take effect, and a warning will be printed in the `dagman.out` file when subsequent *JOBSTATE_LOG* specifications are parsed. Multiple specifications may exist in the same DAG file, within splices, or within multiple, independent DAGs run with a single *condor_dagman* instance.

The `jobstate.log` file can be considered a filtered version of the `dagman.out` file, in a machine-readable format. It contains the actual node job events that from *condor_dagman*, plus some additional meta-events.

The `jobstate.log` file is different from the node status file, in that the `jobstate.log` file is appended to, rather than being overwritten as the DAG runs. Therefore, it contains a history of the DAG, rather than a snapshot of the current state of the DAG.

There are 5 line types in the `jobstate.log` file. Each line begins with a Unix timestamp in the form of seconds since the Epoch. Fields within each line are separated by a single space character.

DAGMan start This line identifies the *condor_dagman* job. The formatting of the line is

timestamp INTERNAL *** DAGMAN_STARTED *dagmanCondorID* ***

The *dagmanCondorID* field is the *condor_dagman* job's `ClusterId` attribute, a period, and the `ProcId` attribute.

DAGMan exit This line identifies the completion of the *condor_dagman* job. The formatting of the line is

timestamp INTERNAL *** DAGMAN_FINISHED *exitCode* ***

The *exitCode* field is value the *condor_dagman* job returns upon exit.

Recovery started If the *condor_dagman* job goes into recovery mode, this meta-event is printed. During recovery mode, events will only be printed in the file if they were not already printed before recovery mode started. The formatting of the line is

timestamp INTERNAL *** RECOVERY_STARTED ***

Recovery finished or Recovery failure At the end of recovery mode, either a RECOVERY_FINISHED or RECOVERY_FAILURE meta-event will be printed, as appropriate.

The formatting of the line is

timestamp INTERNAL *** RECOVERY_FINISHED ***

or

timestamp INTERNAL *** RECOVERY_FAILURE ***

Normal This line is used for all other event and meta-event types. The formatting of the line is

timestamp JobName eventName condorID jobTag - sequenceNumber

The *JobName* is the name given to the node job as defined in the DAG input file with the keyword *JOB*. It identifies the node within the DAG.

The *eventName* is one of the many defined event or meta-events given in the lists below.

The *condorID* field is the job's `ClusterId` attribute, a period, and the `ProcId` attribute. There is no *condorID* assigned yet for some meta-events, such as PRE_SCRIPT_STARTED. For these, the dash character ('-') is printed.

The *jobTag* field is defined for the Pegasus workflow manager. Its usage is generalized to be useful to other workflow managers. Pegasus-managed jobs add a line of the following form to their Condor submit description file:

```
+pegasus_site = "local"
```

This defines the string `local` as the *jobTag* field.

Generalized usage adds a set of 2 commands to the Condor submit description file to define a string as the *jobTag* field:

```
+job_tag_name = "+job_tag_value"
+job_tag_value = "viz"
```

This defines the string `viz` as the *jobTag* field. Without any of these added lines within the Condor submit description file, the dash character ('-') is printed for the *jobTag* field.

The *sequenceNumber* is a monotonically-increasing number that starts at one. It is associated with each attempt at running a node. If a node is retried, it gets a new sequence number; a submit failure does not result in a new sequence number. When a rescue DAG is run, the sequence numbers pick up from where they left off within the previous attempt at running the DAG. Note that this only applies if the rescue DAG is run automatically or with the *-dorescuefrom* command-line option.

Here is an example of a very simple Pegasus `jobstate.log` file, assuming the example *jobTag* field of `local`:

```
1292620511 INTERNAL *** DAGMAN_STARTED 4972.0 ***
1292620523 NodeA PRE_SCRIPT_STARTED - local - 1
1292620523 NodeA PRE_SCRIPT_SUCCESS - local - 1
1292620525 NodeA SUBMIT 4973.0 local - 1
1292620525 NodeA EXECUTE 4973.0 local - 1
1292620526 NodeA JOB_TERMINATED 4973.0 local - 1
1292620526 NodeA JOB_SUCCESS 0 local - 1
1292620526 NodeA POST_SCRIPT_STARTED 4973.0 local - 1
1292620531 NodeA POST_SCRIPT_TERMINATED 4973.0 local - 1
1292620531 NodeA POST_SCRIPT_SUCCESS 4973.0 local - 1
1292620535 INTERNAL *** DAGMAN_FINISHED 0 ***
```

Events defining the eventName field
- SUBMIT
- EXECUTE
- EXECUTABLE_ERROR
- CHECKPOINTED
- JOB_EVICTED
- JOB_TERMINATED
- IMAGE_SIZE
- SHADOW_EXCEPTION
- GENERIC
- JOB_ABORTED
- JOB_SUSPENDED
- JOB_UNSUSPENDED
- JOB_HELD
- JOB_RELEASED
- NODE_EXECUTE
- NODE_TERMINATED
- POST_SCRIPT_TERMINATED
- GLOBUS_SUBMIT
- GLOBUS_SUBMIT_FAILED
- GLOBUS_RESOURCE_UP
- GLOBUS_RESOURCE_DOWN
- REMOTE_ERROR
- JOB_DISCONNECTED
- JOB_RECONNECTED
- JOB_RECONNECT_FAILED
- GRID_RESOURCE_UP
- GRID_RESOURCE_DOWN
- GRID_SUBMIT
- JOB_AD_INFORMATION
- JOB_STATUS_UNKNOWN
- JOB_STATUS_KNOWN
- JOB_STAGE_IN
- JOB_STAGE_OUT

Meta-Events defining the eventName field
- SUBMIT_FAILURE
- JOB_SUCCESS

- JOB_FAILURE
- PRE_SCRIPT_STARTED
- PRE_SCRIPT_SUCCESS
- PRE_SCRIPT_FAILURE
- POST_SCRIPT_STARTED
- POST_SCRIPT_SUCCESS
- POST_SCRIPT_FAILURE
- DAGMAN_STARTED
- DAGMAN_FINISHED
- RECOVERY_STARTED
- RECOVERY_FINISHED
- RECOVERY_FAILURE

2.10.12 Utilizing the Power of DAGMan for Large Numbers of Jobs

Using DAGMan is recommended when submitting large numbers of jobs. The recommendation holds whether the jobs are represented by a DAG due to dependencies, or all the jobs are independent of each other, such as they might be in a parameter sweep. DAGMan offers:

- Throttling to limit the number of submitted jobs at any point in time.
- Retry of jobs that fail. A useful tool when an intermittent error may cause a job to fail or fail to run to completion when attempted at one point in time, but not at another point in time. And, note that what constitutes failure is user-defined.
- Automatic generation of the administrative support that facilitates the rerunning of only jobs that fail.
- The ability to run scripts before and/or after the execution of individual jobs.

Each of these capabilities is described in detail (above) within this manual section about DAGMan. To make effective use of DAGMan, there is no way around reading the appropriate subsections.

To run DAGMan with large numbers of independent jobs, there are generally two ways of organizing and specifying the files that control the jobs. Both ways presume that programs or scripts will generate the files, because the files are either large and repetitive or because there are a large number of similar files to be generated representing the large numbers of jobs. The two file types needed are the DAG input file and the submit description file(s) for the Condor jobs represented. Each of the two ways is presented separately:

A unique submit description file for each of the many jobs. A single DAG input file lists each of the jobs and specifies a distinct Condor submit description file for each job. The DAG

input file is simple to generate, as it chooses an identifier for each job and names the submit description file. For example, the simplest DAG input file for a set of 1000 independent jobs, as might be part of a parameter sweep, appears as

```
# file sweep.dag
JOB job0 job0.submit
JOB job1 job1.submit
JOB job2 job2.submit
.
.
.
JOB job999 job999.submit
```

There are 1000 submit description files, with a unique one for each of the job<N> jobs. Assuming that all files associated with this set of jobs are in the same directory, and that files continue the same naming and numbering scheme, the submit description file for `job6.submit` might appear as

```
# file job6.submit
universe = vanilla
executable = /path/to/executable
log = job6.log
input = job6.in
output = job6.out
notification = Never
arguments = "-file job6.out"
queue
```

Submission of the entire set of jobs is

```
condor_submit_dag sweep.dag
```

A benefit to having unique submit description files for each of the jobs is that they are available, if one of the jobs needs to be submitted individually. A drawback to having unique submit description files for each of the jobs is that there are lots of files, one for each job.

Single submit description file. A single Condor submit description file might be used for all the many jobs of the parameter sweep. To distinguish the jobs and their associated distinct input and output files, the DAG input file assigns a unique identifier with the *VARS* keyword.

```
# file sweep.dag
JOB job0 common.submit
VARS job0 runnumber="0"
JOB job1 common.submit
VARS job1 runnumber="1"
JOB job2 common.submit
VARS job2 runnumber="2"
```

```
.
.
.
JOB job999 common.submit
VARS job999 runnumber="999"
```

The single submit description file for all these jobs utilizes the `runnumber` variable value in its identification of the job's files. This submit description file might appear as

```
# file common.submit
universe = vanilla
executable = /path/to/executable
log = job$(runnumber).log
input = job$(runnumber).in
output = job$(runnumber).out
notification = Never
arguments = "-$(runnumber)"
queue
```

The job with `runnumber="8"` expects to find its input file `job8.in` in the single, common directory, and it logs job events in `job8.log` and sends its output to `job8.out`. The executable is invoked with

```
/path/to/executable -8
```

These examples work well with respect to file naming and placement when there are less than several thousand jobs submitted as part of a DAG. The large numbers of files per directory becomes an issue when there are greater than several thousand jobs submitted as part of a DAG. In this case, consider a more hierarchical structure for the files instead of a single directory. Introduce a separate directory for each run. For example, if there were 10,000 jobs, there would be 10,000 directories, one for each of these jobs. The directories are presumed to be generated and populated by programs or scripts that, like the previous examples, utilize a run number. Each of these directories named utilizing the run number will be used for the input, output, and log files for one of the many jobs.

As an example, for this set of 10,000 jobs and directories, assume that there is a run number of 600. The directory will be named `dir.600`, and it will hold the 3 files called `in`, `out`, and `log`, representing the input, output, and Condor job log files associated with run number 600.

The DAG input file sets a variable representing the run number, as in the previous example:

```
# file biggersweep.dag
JOB job0 common.submit
VARS job0 runnumber="0"
JOB job1 common.submit
VARS job1 runnumber="1"
JOB job2 common.submit
VARS job2 runnumber="2"
```

```
.
.
.
JOB job9999 common.submit
VARS job9999 runnumber="9999"
```

A single Condor submit description file may be written. It resides in the same directory as the DAG input file.

```
# file bigger.submit
universe = vanilla
executable = /path/to/executable
log = log
input = in
output = out
notification = Never
arguments = "-$(runnumber)"
initialdir = dir.$(runnumber)
queue
```

One item to care about with this set up is the underlying file system for the pool. The transfer of files (or not) when using **initialdir** differs based upon the job **universe** and whether or not there is a shared file system. See section 9 for the details on the submit command **initialdir**.

Submission of this set of jobs is no different than the previous examples. With the current working directory the same as the one containing the submit description file, the DAG input file, and the subdirectories,

```
condor_submit_dag biggersweep.dag
```

2.11 Virtual Machine Applications

The **vm** universe facilitates a Condor job that matches and then lands a disk image on an execute machine within a Condor pool. This disk image is intended to be a virtual machine. In this manner, the virtual machine is the job to be executed.

This section describes this type of Condor job. See section 3.3.29 for details of configuration variables.

2.11.1 The Submit Description File

Different than all other universe jobs, the **vm** universe job specifies a disk image, not an executable. Therefore, the submit commands **input**, **output**, and **error** do not apply. If specified, *condor_submit*

rejects the job with an error. The **executable** command changes definition within a **vm** universe job. It no longer specifies an executable file, but instead provides a string that identifies the job for tools such as *condor_q*. Other commands specific to the type of virtual machine software identify the disk image.

VMware, Xen, and KVM virtual machine software are supported. As these differ from each other, the submit description file specifies one of

```
vm_type = vmware
```

or

```
vm_type = xen
```

or

```
vm_type = kvm
```

The job is required to specify its memory needs for the disk image with **vm_memory**, which is given in Mbytes. Condor uses this number to assure a match with a machine that can provide the needed memory space.

Virtual machine networking is enabled with the command

```
vm_networking = true
```

And, when networking is enabled, a definition of **vm_networking_type** as **bridge** matches the job only with a machine that is configured to use bridge networking. A definition of **vm_networking_type** as **nat** matches the job only with a machine that is configured to use NAT networking. When no definition of **vm_networking_type** is given, Condor may match the job with a machine that enables networking, and further, the choice of bridge or NAT networking is determined by the machine's configuration.

Modified disk images are transferred back to the machine from which the job was submitted as the **vm** universe job completes. Job completion for a **vm** universe job occurs when the virtual machine is shut down, and Condor notices (as the result of a periodic check on the state of the virtual machine). Should the job not want any files transferred back (modified or not), for example because the job explicitly transferred its own files, the submit command to prevent the transfer is

```
vm_no_output_vm = true
```

The required disk image must be identified for a virtual machine. This **vm_disk** command specifies a list of comma-separated files. Each disk file is specified by 4 colon separated fields. The first field is the path and file name of the disk file. The second field specifies the device, The third field specifies permissions, and the optional fourth specifies the format. Here is an example that identifies two files:

```
vm_disk = /var/lib/libvirt/images/swap.img:sda2:w:raw
```

Setting values in the submit description file for some commands have consequences for the virtual machine description file. These commands are

- **vm_memory**
- **vm_macaddr**
- **vm_networking**
- **vm_networking_type**
- **vm_disk**

For VMware virtual machines, setting values for these commands causes Condor to modify the `.vmx` file, overwriting existing values. For KVM and Xen virtual machines, Condor uses these values when it produces the description file.

Further commands specify information that is specific to the virtual machine type targeted.

VMware-Specific Submit Commands

Specific to VMware, the submit description file command **vmware_dir** gives the path and directory (on the machine from which the job is submitted) where VMware-specific files and applications reside. One example of a VMware-specific application is the VMDK files, which form a virtual hard drive (disk image) for the virtual machine. VMX files containing the primary configuration for the virtual machine would also be in this directory.

Condor must be told whether or not the contents of the **vmware_dir** directory must be transferred to the machine where the job is to be executed. This required information is given with the submit command **vmware_should_transfer_files**. With a value of `True`, Condor does transfer the contents of the directory. With a value of `False`, Condor does not transfer the contents of the directory, and instead presumes that access to this directory is available through a shared file system.

By default, Condor uses a snapshot disk for new and modified files. They may also be utilized for checkpoints. The snapshot disk is initially quite small, growing only as new files are created or files are modified. When **vmware_should_transfer_files** is `True`, a job may specify that a snapshot disk is *not* to be used with the command

```
vmware_snapshot_disk = False
```

In this case, Condor will utilize original disk files in producing checkpoints. Note that *condor_submit* issues an error message and does not submit the job if both **vmware_should_transfer_files** and **vmware_snapshot_disk** are `False`.

Note that if snapshot disks are requested and file transfer is not being used, the **vmware_dir** setting given in the submit description file should not contain any symbolic link path components. This is to work around the issue discussed in the FAQ entry in section 7.3.

Here is a sample submit description file for a VMware virtual machine:

```
universe                       = vm
executable                     = vmware_sample_job
log                            = simple.vm.log.txt
vm_type                        = vmware
vm_memory                      = 64
vmware_dir                     = C:\condor-test
vmware_should_transfer_files = True
queue
```

This sample uses the **vmware_dir** command to identify the location of the disk image to be executed as a Condor job. The contents of this directory are transferred to the machine assigned to execute the Condor job.

Xen-Specific Submit Commands

If any files need to be transferred from the submit machine to the machine where the **vm** universe job will execute, Condor must be explicitly told to do so with the **xen_transfer_files** command:

```
xen_transfer_files = /myxen/diskfile.img,/myxen/swap.img
```

Any and all needed files on a system without a shared file system (between the submit machine and the machine where the job will execute) must be listed.

A Xen **vm** universe job requires specification of the guest kernel. The **xen_kernel** command accomplishes this, utilizing one of the following definitions.

1. **xen_kernel = included** implies that the kernel is to be found in disk image given by the definition of the single file specified in **vm_disk**.
2. **xen_kernel = path-to-kernel** gives a full path and file name of the required kernel. If this kernel must be transferred to machine on which the **vm** universe job will execute, it must also be included in the **xen_transfer_files** command.

   This form of the **xen_kernel** command also requires further definition of the **xen_root** command. **xen_root** defines the device containing files needed by `root`.

KVM-Specific Submit Commands

If any files need to be transferred from the submit machine to the machine where the **vm** universe job will execute, Condor must be explicitly told to do so with the **kvm_transfer_files** command:

```
kvm_transfer_files = /var/lib/libvirt/images/swap.img
```

Any and all needed files on a system without a shared file system (between the submit machine and the machine where the job will execute) must be listed.

2.11.2 Checkpoints

Creating a checkpoint is straightforward for a virtual machine, as a checkpoint is a set of files that represent a snapshot of both disk image and memory. The checkpoint is created and all files are transferred back to the `$(SPOOL)` directory on the machine from which the job was submitted. The submit command to create checkpoints is

```
vm_checkpoint = true
```

Without this command, no checkpoints are created (by default). With the command, a checkpoint is created any time the **vm** universe jobs is evicted from the machine upon which it is executing. This occurs as a result of the machine configuration indicating that it will no longer execute this job.

vm universe jobs can *not* use a checkpoint server.

Periodically creating checkpoints is not supported at this time.

Enabling both networking and checkpointing for a **vm** universe job can cause networking problems when the job restarts, particularly if the job migrates to a different machine. *condor_submit* will normally reject such jobs. To enable both, then add the command

```
when_to_transfer_output = ON_EXIT_OR_EVICT
```

Take care with respect to the use of network connections within the virtual machine and their interaction with checkpoints. Open network connections at the time of the checkpoint will likely be lost when the checkpoint is subsequently used to resume execution of the virtual machine. This occurs whether or not the execution resumes on the same machine or a different one within the Condor pool.

2.11.3 Disk Images

VMware on Windows and Linux

Following the platform-specific guest OS installation instructions found at http://pubs.vmware.com/guestnotes, creates a VMware disk image.

Xen and KVM

While the following web page contains instructions specific to Fedora on how to create a virtual guest image, it should provide a good starting point for other platforms as well.

http://fedoraproject.org/wiki/Virtualization_Quick_Start

2.11.4 Job Completion in the vm Universe

Job completion for a **vm** universe job occurs when the virtual machine is shut down, and Condor notices (as the result of a periodic check on the state of the virtual machine). This is different from jobs executed under the environment of other universes.

Shut down of a virtual machine occurs from within the virtual machine environment. A script, executed with the proper authorization level, is the likely source of the shut down commands.

Under a Windows 2000, Windows XP, or Vista virtual machine, an administrator issues the command

```
shutdown -s -t 01
```

For older versions of Windows operating systems, directions are given at http://www.aumha.org/win4/a/shutcut.php.

Under a Linux virtual machine, the `root` user executes

```
/sbin/poweroff
```

The command `/sbin/halt` will not completely shut down some Linux distributions, and instead causes the job to hang.

Since the successful completion of the **vm** universe job requires the successful shut down of the virtual machine, it is good advice to try the shut down procedure outside of Condor, before a **vm** universe job is submitted.

2.12 Time Scheduling for Job Execution

Jobs may be scheduled to begin execution at a specified time in the future with Condor's job deferral functionality. All specifications are in a job's submit description file. Job deferral functionality is expanded to provide for the periodic execution of a job, known as the CronTab scheduling.

2.12.1 Job Deferral

Job deferral allows the specification of the exact date and time at which a job is to begin executing. Condor attempts to match the job to an execution machine just like any other job, however, the job will wait until the exact time to begin execution. A user can specify Condor to allow some flexibility to execute jobs that miss their execution time.

Deferred Execution Time

A job's deferral time is the exact time that Condor should attempt to execute the job. The deferral time attribute is defined as an expression that evaluates to a Unix Epoch timestamp (the number of seconds elapsed since 00:00:00 on January 1, 1970, Coordinated Universal Time). This is the time that Condor will begin to execute the job.

After a job is matched and all of its files have been transferred to an execution machine, Condor checks to see if the job's ad contains a deferral time. If it does, Condor calculates the number of seconds between the execution machine's current system time to the job's deferral time. If the deferral time is in the future, the job waits to begin execution. While a job waits, its job ClassAd attribute `JobStatus` indicates the job is running. As the deferral time arrives, the job begins to execute. If a job misses its execution time, that is, if the deferral time is in the past, the job is evicted from the execution machine and put on hold in the queue.

The specification of a deferral time does not interfere with Condor's behavior. For example, if a job is waiting to begin execution when a *condor_hold* command is issued, the job is removed from the execution machine and is put on hold. If a job is waiting to begin execution when a *condor_suspend* command is issued, the job continues to wait. When the deferral time arrives, Condor begins execution for the job, but immediately suspends it.

Missed Execution Window

If a job arrives at its execution machine after the deferral time passes, the job is evicted from the machine and put on hold in the job queue. This may occur, for example, because the transfer of needed files took too long due to a slow network connection. A deferral window permits the execution of a job that misses its deferral time by specifying a window of time within which the job may begin.

The deferral window is the number of seconds after the deferral time, within which the job may begin. When a job arrives too late, Condor calculates the difference in seconds between the execution machine's current time and the job's deferral time. If this difference is less than or equal to the deferral window, the job immediately begins execution. If this difference is greater than the deferral window, the job is evicted from the execution machine and is put on hold in the job queue.

Preparation Time

When a job defines a deferral time far in the future and then is matched to an execution machine, potential computation cycles are lost because the deferred job has claimed the machine, but is not actually executing. Other jobs could execute during the interval when the job waits for its deferral time. To make use of the wasted time, a job defines a **deferral_prep_time** with an integer expression that evaluates to a number of seconds. At this number of seconds before the deferral time, the job may be matched with a machine.

Usage Examples

Here are examples of how the job deferral time, deferral window, and the preparation time may be used.

The job's submit description file specifies that the job is to begin execution on January 1st, 2006 at 12:00 pm:

```
deferral_time = 1136138400
```

The Unix *date* program may be used to calculate a Unix epoch time. The syntax of the command to do this depends on the options provided within that flavor of Unix. In some, it appears as

```
%  date --date "MM/DD/YYYY HH:MM:SS" +%s
```

and in others, it appears as

```
%  date -d "YYYY-MM-DD HH:MM:SS" +%s
```

`MM` is a 2-digit month number, `DD` is a 2-digit day of the month number, and `YYYY` is a 4-digit year. `HH` is the 2-digit hour of the day, `MM` is the 2-digit minute of the hour, and `SS` are the 2-digit seconds within the minute. The characters `+%s` tell the *date* program to give the output as a Unix epoch time.

The job always waits 60 seconds before beginning execution:

```
deferral_time = (CurrentTime + 60)
```

In this example, assume that the deferral time is 45 seconds in the past as the job is available. The job begins execution, because 75 seconds remain in the deferral window:

```
deferral_window = 120
```

In this example, a job is scheduled to execute far in the future, on January 1st, 2010 at 12:00 pm. The **deferral_prep_time** attribute delays the job from being matched until 60 seconds before the job is to begin execution.

```
deferral_time      = 1262368800
deferral_prep_time = 60
```

Limitations

There are some limitations to Condor's job deferral feature.

- Job deferral is not available for scheduler universe jobs. A scheduler universe job defining the `deferral_time` produces a fatal error when submitted.
- The time that the job begins to execute is based on the execution machine's system clock, and not the submission machine's system clock. Be mindful of the ramifications when the two clocks show dramatically different times.
- A job's `JobStatus` attribute is always in the running state when job deferral is used. There is currently no way to distinguish between a job that is executing and a job that is waiting for its deferral time.

2.12.2 CronTab Scheduling

Condor's CronTab scheduling functionality allows jobs to be scheduled to execute periodically. A job's execution schedule is defined by commands within the submit description file. The notation is much like that used by the Unix *cron* daemon. As such, Condor developers are fond of referring to CronTab scheduling as *Crondor*. The scheduling of jobs using Condor's CronTab feature calculates and utilizes the `DeferralTime` ClassAd attribute.

Also, unlike the Unix *cron* daemon, Condor never runs more than one instance of a job at the same time.

The capability for repetitive or periodic execution of the job is enabled by specifying an **on_exit_remove** command for the job, such that the job does not leave the queue until desired.

Semantics for CronTab Specification

A job's execution schedule is defined by a set of specifications within the submit description file. Condor uses these to calculate a `DeferralTime` for the job.

Table 2.2 lists the submit commands and acceptable values for these commands. At least one of these must be defined in order for Condor to calculate a `DeferralTime` for the job. Once

Submit Command	Allowed Values
cron_minute	0 - 59
cron_hour	0 - 23
cron_day_of_month	1 - 31
cron_month	1 - 12
cron_day_of_week	0 - 7 (Sunday is 0 or 7)

Table 2.2: The list of submit commands and their value ranges.

one CronTab value is defined, the default for all the others uses all the values in the allowed values ranges.

The day of a job's execution can be specified by both the **cron_day_of_month** and the **cron_day_of_week** attributes. The day will be the logical or of both.

The semantics allow more than one value to be specified by using the `*` operator, ranges, lists, and steps (strides) within ranges.

The asterisk operator The `*` (asterisk) operator specifies that all of the allowed values are used for scheduling. For example,

```
cron_month = *
```

becomes any and all of the list of possible months: (1,2,3,4,5,6,7,8,9,10,11,12). Thus, a job runs any month in the year.

Ranges A range creates a set of integers from all the allowed values between two integers separated by a hyphen. The specified range is inclusive, and the integer to the left of the hyphen must be less than the right hand integer. For example,

```
cron_hour = 0-4
```

represents the set of hours from 12:00 am (midnight) to 4:00 am, or (0,1,2,3,4).

Lists A list is the union of the values or ranges separated by commas. Multiple entries of the same value are ignored. For example,

```
cron_minute = 15,20,25,30
cron_hour   = 0-3,9-12,15
```

cron_minute represents (15,20,25,30) and **cron_hour** represents (0,1,2,3,9,10,11,12,15).

Steps Steps select specific numbers from a range, based on an interval. A step is specified by appending a range or the asterisk operator with a slash character (`/`), followed by an integer value. For example,

```
cron_minute = 10-30/5
cron_hour = */3
```

cron_minute specifies every five minutes within the specified range to represent (10,15,20,25,30). **cron_hour** specifies every three hours of the day to represent (0,3,6,9,12,15,18,21).

Preparation Time and Execution Window

The **cron_prep_time** command is analogous to the deferral time's **deferral_prep_time** command. It specifies the number of seconds before the deferral time that the job is to be matched and sent to the execution machine. This permits Condor to make necessary preparations before the deferral time occurs.

Consider the submit description file example that includes

```
cron_minute = 0
cron_hour = *
cron_prep_time = 300
```

The job is scheduled to begin execution at the top of every hour. Note that the setting of **cron_hour** in this example is not required, as the default value will be `*`, specifying any and every hour of the day. The job will be matched and sent to an execution machine no more than five minutes before the next deferral time. For example, if a job is submitted at 9:30am, then the next deferral time will be calculated to be 10:00am. Condor may attempt to match the job to a machine and send the job once it is 9:55am.

As the CronTab scheduling calculates and uses deferral time, jobs may also make use of the deferral window. The submit command **cron_window** is analogous to the submit command **deferral_window**. Consider the submit description file example that includes

```
cron_minute = 0
cron_hour = *
cron_window = 360
```

As the previous example, the job is scheduled to begin execution at the top of every hour. Yet with no preparation time, the job is likely to miss its deferral time. The 6-minute window allows the job to begin execution, as long as it arrives and can begin within 6 minutes of the deferral time, as seen by the time kept on the execution machine.

Scheduling

When a job using the CronTab functionality is submitted to Condor, use of at least one of the submit description file commands beginning with `cron_` causes Condor to calculate and set a deferral time

for when the job should run. A deferral time is determined based on the current time rounded later in time to the next minute. The deferral time is the job's `DeferralTime` attribute. A new deferral time is calculated when the job first enters the job queue, when the job is re-queued, or when the job is released from the hold state. New deferral times for *all* jobs in the job queue using the CronTab functionality are recalculated when a *condor_reconfig* or a *condor_restart* command that affects the job queue is issued.

A job's deferral time is not always the same time that a job will receive a match and be sent to the execution machine. This is because Condor operates on the job queue at times that are independent of job events, such as when job execution completes. Therefore, Condor may operate on the job queue just after a job's deferral time states that it is to begin execution. Condor attempts to start a job when the following pseudo-code boolean expression evaluates to `True`:

```
( CurrentTime + SCHEDD_INTERVAL ) >= ( DeferralTime - CronPrepTime )
```

If the `CurrentTime` plus the number of seconds until the next time Condor checks the job queue is greater than or equal to the time that the job should be submitted to the execution machine, then the job is to be matched and sent now.

Jobs using the CronTab functionality are not automatically re-queued by Condor after their execution is complete. The submit description file for a job must specify an appropriate **on_exit_remove** command to ensure that a job remains in the queue. This job maintains its original `ClusterId` and `ProcId`.

Usage Examples

Here are some examples of the submit commands necessary to schedule jobs to run at multifarious times. Please note that it is not necessary to explicitly define each attribute; the default value is `*`.

Run 23 minutes after every two hours, every day of the week:

```
on_exit_remove = false
cron_minute = 23
cron_hour = 0-23/2
cron_day_of_month = *
cron_month = *
cron_day_of_week = *
```

Run at 10:30pm on each of May 10th to May 20th, as well as every remaining Monday within the month of May:

```
on_exit_remove = false
cron_minute = 30
cron_hour = 20
```

```
cron_day_of_month = 10-20
cron_month = 5
cron_day_of_week = 2
```

Run on every 10 minutes and every 6 minutes before noon on January 18th with a 2-minute preparation time:

```
on_exit_remove = false
cron_minute = */10,*/6
cron_hour = 0-11
cron_day_of_month = 18
cron_month = 1
cron_day_of_week = *
cron_prep_time = 120
```

Limitations

The use of the CronTab functionality has all of the same limitations of deferral times, because the mechanism is based upon deferral times.

- It is impossible to schedule vanilla and standard universe jobs at intervals that are smaller than the interval at which Condor evaluates jobs. This interval is determined by the configuration variable `SCHEDD_INTERVAL` . As a vanilla or standard universe job completes execution and is placed back into the job queue, it may not be placed in the idle state in time. This problem does not afflict local universe jobs.
- Condor cannot guarantee that a job will be matched in order to make its scheduled deferral time. A job must be matched with an execution machine just as any other Condor job; if Condor is unable to find a match, then the job will miss its chance for executing and must wait for the next execution time specified by the CronTab schedule.

2.13 Job Monitor

The Condor Job Monitor is a Java application designed to allow users to view user log files.

To view a user log file, select it using the open file command in the File menu. After the file is parsed, it will be visually represented. Each horizontal line represents an individual job. The x-axis is time. Whether a job is running at a particular time is represented by its color at that time – white for running, black for idle. For example, a job which appears predominantly white has made efficient progress, whereas a job which appears predominantly black has received an inordinately small proportion of computational time.

2.13.1 Transition States

A transition state is the state of a job at any time. It is called a "transition" because it is defined by the two events which bookmark it. There are two basic transition states: running and idle. An idle job typically is a job which has just been submitted into the Condor pool and is waiting to be matched with an appropriate machine or a job which has vacated from a machine and has been returned to the pool. A running job, by contrast, is a job which is making active progress.

Advanced users may want a visual distinction between two types of running transitions: "goodput" or "badput". Goodput is the transition state preceding an eventual job completion or checkpoint. Badput is the transition state preceding a non-checkpointed eviction event. Note that "badput" is potentially a misleading nomenclature; a job which is not checkpointed by the Condor program may checkpoint itself or make progress in some other way. To view these two transition as distinct transitions, select the appropriate option from the "View" menu.

2.13.2 Events

There are two basic kinds of events: checkpoint events and error events. Plus advanced users can ask to see more events.

2.13.3 Selecting Jobs

To view any arbitrary selection of jobs in a job file, use the job selector tool. Jobs appear visually by order of appearance within the actual text log file. For example, the log file might contain jobs 775.1, 775.2, 775.3, 775.4, and 775.5, which appear in that order. A user who wishes to see only jobs 775.2 and 775.5 can select only these two jobs in the job selector tool and click the "Ok" or "Apply" button. The job selector supports double clicking; double click on any single job to see it drawn in isolation.

2.13.4 Zooming

To view a small area of the log file, zoom in on the area which you would like to see in greater detail. You can zoom in, out and do a full zoom. A full zoom redraws the log file in its entirety. For example, if you have zoomed in very close and would like to go all the way back out, you could do so with a succession of zoom outs or with one full zoom.

There is a difference between using the menu driven zooming and the mouse driven zooming. The menu driven zooming will recenter itself around the current center, whereas mouse driven zooming will recenter itself (as much as possible) around the mouse click. To help you re-find the clicked area, a box will flash after the zoom. This is called the "zoom finder" and it can be turned off in the zoom menu if you prefer.

2.13.5 Keyboard and Mouse Shortcuts

1. The Keyboard shortcuts:
    - Arrows - an approximate ten percent scrollbar movement
    - PageUp and PageDown - an approximate one hundred percent scrollbar movement
    - Control + Left or Right - approximate one hundred percent scrollbar movement
    - End and Home - scrollbar movement to the vertical extreme
    - Others - as seen beside menu items
2. The mouse shortcuts:
    - Control + Left click - zoom in
    - Control + Right click - zoom out
    - Shift + left click - re-center

2.14 Special Environment Considerations

2.14.1 AFS

The Condor daemons do not run authenticated to AFS; they do not possess AFS tokens. Therefore, no child process of Condor will be AFS authenticated. The implication of this is that you must set file permissions so that your job can access any necessary files residing on an AFS volume without relying on having your AFS permissions.

If a job you submit to Condor needs to access files residing in AFS, you have the following choices:

1. Copy the needed files from AFS to either a local hard disk where Condor can access them using remote system calls (if this is a standard universe job), or copy them to an NFS volume.
2. If the files must be kept on AFS, then set a host ACL (using the AFS *fs setacl* command) on the subdirectory to serve as the current working directory for the job. If this is a standard universe job, then the host ACL needs to give read/write permission to any process on the submit machine. If this is a vanilla universe job, then set the ACL such that any host in the pool can access the files without being authenticated. If you do not know how to use an AFS host ACL, ask the person at your site responsible for the AFS configuration.

The Condor Team hopes to improve upon how Condor deals with AFS authentication in a subsequent release.

Please see section 3.13.1 on page 432 in the Administrators Manual for further discussion of this problem.

2.14.2 NFS

If the current working directory when a job is submitted, as with *condor_submit*, is accessed via an NFS automounter, Condor may have problems if the automounter later decides to unmount the volume before the job has completed. This is because *condor_submit* likely has stored the dynamic mount point as the job's initial current working directory, and this mount point could become automatically unmounted by the automounter.

There is a simple work around. When submitting the job, use the *initialdir* command in the submit description file to point to the stable access point. For example, suppose the NFS automounter is configured to mount a volume at mount point `/a/myserver.company.com/vol1/johndoe` whenever the directory `/home/johndoe` is accessed. Adding the following line to the submit description file solves the problem.

```
initialdir = /home/johndoe
```

As of Condor version 7.4.0, Condor attempts to flush the NFS cache on a submit machine in order to refresh a job's initial working directory. This allows files written by the job into an NFS mounted initial working directory to be immediately visible on the submit machine. Since the flush operation can require multiple round trips to the NFS server, it is expensive. Therefore, a job may disable the flushing by setting

```
+IwdFlushNFSCache = False
```

in the job's submit description file. See page 905 for a definition of the job ClassAd attribute.

2.14.3 Condor Daemons That Do Not Run as root

Condor is normally installed such that the Condor daemons have root permission. This allows Condor to run the condor_shadow process and your job with your UID and file access rights. When Condor is started as root, your Condor jobs can access whatever files you can.

However, it is possible that whomever installed Condor did not have root access, or decided not to run the daemons as root. That is unfortunate, since Condor is designed to be run as the Unix user root. To see if Condor is running as root on a specific machine, enter the command

```
condor_status -master -l <machine-name>
```

where `machine-name` is the name of the specified machine. This command displays a condor_master ClassAd; if the attribute `RealUid` equals zero, then the Condor daemons are indeed running with root access. If the `RealUid` attribute is not zero, then the Condor daemons do not have root access.

NOTE: The Unix program *ps* is *not* an effective method of determining if Condor is running with root access. When using *ps*, it may often appear that the daemons are running as the condor

user instead of root. However, note that the *ps,* command shows the current *effective* owner of the process, not the *real* owner. (See the *getuid*(2) and *geteuid*(2) Unix man pages for details.) In Unix, a process running under the real UID of root may switch its effective UID. (See the *seteuid*(2) man page.) For security reasons, the daemons only set the effective UID to root when absolutely necessary (to perform a privileged operation).

If they are not running with root access, you need to make any/all files and/or directories that your job will touch readable and/or writable by the UID (user id) specified by the RealUid attribute. Often this may mean using the Unix command `chmod 777` on the directory where you submit your Condor job.

2.14.4 Job Leases

A job lease specifies how long a given job will attempt to run on a remote resource, even if that resource loses contact with the submitting machine. Similarly, it is the length of time the submitting machine will spend trying to reconnect to the (now disconnected) execution host, before the submitting machine gives up and tries to claim another resource to run the job. The goal aims at run only once semantics, so that the *condor_schedd* daemon does not allow the same job to run on multiple sites simultaneously.

If the submitting machine is alive, it periodically renews the job lease, and all is well. If the submitting machine is dead, or the network goes down, the job lease will no longer be renewed. Eventually the lease expires. While the lease has not expired, the execute host continues to try to run the job, in the hope that the submit machine will come back to life and reconnect. If the job completes and the lease has not expired, yet the submitting machine is still dead, the *condor_starter* daemon will wait for a *condor_shadow* daemon to reconnect, before sending final information on the job, and its output files. Should the lease expire, the *condor_startd* daemon kills off the *condor_starter* daemon and user job.

A default value equal to 20 minutes exists for a job's ClassAd attribute `job_lease_duration`, or this attribute may be set in the submit description file to keep a job running in the case that the submit side no longer renews the lease. There is a trade off in setting the value of `job_lease_duration`. Too small a value, and the job might get killed before the submitting machine has a chance to recover. Forward progress on the job will be lost. Too large a value, and an execute resource will be tied up waiting for the job lease to expire. The value should be chosen based on how long the user is willing to tie up the execute machines, how quickly submit machines come back up, and how much work would be lost if the lease expires, the job is killed, and the job must start over from its beginning.

As a special case, a submit description file setting of

```
job_lease_duration = 0
```

as well as utilizing submission other than *condor_submit* that do not set `JobLeaseDuration` (such as using the web services interface) results in the corresponding job ClassAd attribute to be

explicitly undefined. This has the further effect of changing the duration of a claim lease, the amount of time that the execution machine waits before dropping a claim due to missing keep alive messages.

2.15 Potential Problems

2.15.1 Renaming of argv[0]

When Condor starts up your job, it renames argv[0] (which usually contains the name of the program) to condor_exec. This is convenient when examining a machine's processes with the Unix command *ps*; the process is easily identified as a Condor job.

Unfortunately, some programs read argv[0] expecting their own program name and get confused if they find something unexpected like condor_exec.

CHAPTER

THREE

Administrators' Manual

3.1 Introduction

This is the Condor Administrator's Manual for Unix. Its purpose is to aid in the installation and administration of a Condor pool. For help on using Condor, see the Condor User's Manual.

A Condor pool is comprised of a single machine which serves as the *central manager*, and an arbitrary number of other machines that have joined the pool. Conceptually, the pool is a collection of resources (machines) and resource requests (jobs). The role of Condor is to match waiting requests with available resources. Every part of Condor sends periodic updates to the central manager, the centralized repository of information about the state of the pool. Periodically, the central manager assesses the current state of the pool and tries to match pending requests with the appropriate resources.

Each resource has an owner, the user who works at the machine. This person has absolute power over their own resource and Condor goes out of its way to minimize the impact on this owner caused by Condor. It is up to the resource owner to define a policy for when Condor requests will serviced and when they will be denied.

Each resource request has an owner as well: the user who submitted the job. These people want Condor to provide as many CPU cycles as possible for their work. Often the interests of the resource owners are in conflict with the interests of the resource requesters.

The job of the Condor administrator is to configure the Condor pool to find the happy medium that keeps both resource owners and users of resources satisfied. The purpose of this manual is to help you understand the mechanisms that Condor provides to enable you to find this happy medium for your particular set of users and resource owners.

3.1.1 The Different Roles a Machine Can Play

Every machine in a Condor pool can serve a variety of roles. Most machines serve more than one role simultaneously. Certain roles can only be performed by single machines in your pool. The following list describes what these roles are and what resources are required on the machine that is providing that service:

Central Manager There can be only one central manager for your pool. The machine is the collector of information, and the negotiator between resources and resource requests. These two halves of the central manager's responsibility are performed by separate daemons, so it would be possible to have different machines providing those two services. However, normally they both live on the same machine. This machine plays a very important part in the Condor pool and should be reliable. If this machine crashes, no further matchmaking can be performed within the Condor system (although all current matches remain in effect until they are broken by either party involved in the match). Therefore, choose for central manager a machine that is likely to be up and running all the time, or at least one that will be rebooted quickly if something goes wrong. The central manager will ideally have a good network connection to all the machines in your pool, since they all send updates over the network to the central manager. All queries go to the central manager.

Execute Any machine in your pool (including your Central Manager) can be configured for whether or not it should execute Condor jobs. Obviously, some of your machines will have to serve this function or your pool won't be very useful. Being an execute machine doesn't require many resources at all. About the only resource that might matter is disk space, since if the remote job dumps core, that file is first dumped to the local disk of the execute machine before being sent back to the submit machine for the owner of the job. However, if there isn't much disk space, Condor will simply limit the size of the core file that a remote job will drop. In general the more resources a machine has (swap space, real memory, CPU speed, etc.) the larger the resource requests it can serve. However, if there are requests that don't require many resources, any machine in your pool could serve them.

Submit Any machine in your pool (including your Central Manager) can be configured for whether or not it should allow Condor jobs to be submitted. The resource requirements for a submit machine are actually much greater than the resource requirements for an execute machine. First of all, every job that you submit that is currently running on a remote machine generates another process on your submit machine. So, if you have lots of jobs running, you will need a fair amount of swap space and/or real memory. In addition all the checkpoint files from your jobs are stored on the local disk of the machine you submit from. Therefore, if your jobs have a large memory image and you submit a lot of them, you will need a lot of disk space to hold these files. This disk space requirement can be somewhat alleviated with a checkpoint server (described below), however the binaries of the jobs you submit are still stored on the submit machine.

Checkpoint Server One machine in your pool can be configured as a checkpoint server. This is optional, and is not part of the standard Condor binary distribution. The checkpoint server is a centralized machine that stores all the checkpoint files for the jobs submitted in your pool.

This machine should have lots of disk space and a good network connection to the rest of your pool, as the traffic can be quite heavy.

Now that you know the various roles a machine can play in a Condor pool, we will describe the actual daemons within Condor that implement these functions.

3.1.2 The Condor Daemons

The following list describes all the daemons and programs that could be started under Condor and what they do:

condor_master This daemon is responsible for keeping all the rest of the Condor daemons running on each machine in your pool. It spawns the other daemons, and periodically checks to see if there are new binaries installed for any of them. If there are, the master will restart the affected daemons. In addition, if any daemon crashes, the master will send e-mail to the Condor Administrator of your pool and restart the daemon. The *condor_master* also supports various administrative commands that let you start, stop or reconfigure daemons remotely. The *condor_master* will run on every machine in your Condor pool, regardless of what functions each machine are performing.

condor_startd This daemon represents a given resource (namely, a machine capable of running jobs) to the Condor pool. It advertises certain attributes about that resource that are used to match it with pending resource requests. The startd will run on any machine in your pool that you wish to be able to execute jobs. It is responsible for enforcing the policy that resource owners configure which determines under what conditions remote jobs will be started, suspended, resumed, vacated, or killed. When the startd is ready to execute a Condor job, it spawns the *condor_starter*, described below.

condor_starter This program is the entity that actually spawns the remote Condor job on a given machine. It sets up the execution environment and monitors the job once it is running. When a job completes, the starter notices this, sends back any status information to the submitting machine, and exits.

condor_schedd This daemon represents resource requests to the Condor pool. Any machine that you wish to allow users to submit jobs from needs to have a *condor_schedd* running. When users submit jobs, they go to the schedd, where they are stored in the *job queue*, which the schedd manages. Various tools to view and manipulate the job queue (such as *condor_submit*, *condor_q*, or *condor_rm*) all must connect to the schedd to do their work. If the schedd is down on a given machine, none of these commands will work.

The *condor_schedd* advertises the number of waiting jobs in its job queue and is responsible for claiming available resources to serve those requests. Once a schedd has been matched with a given resource, the schedd spawns a *condor_shadow* (described below) to serve that particular request.

condor_shadow This program runs on the machine where a given request was submitted and acts as the resource manager for the request. Jobs that are linked for Condor's standard universe, which perform remote system calls, do so via the *condor_shadow*. Any system call performed on the remote execute machine is sent over the network, back to the *condor_shadow* which actually performs the system call (such as file I/O) on the submit machine, and the result is sent back over the network to the remote job. In addition, the shadow is responsible for making decisions about the request (such as where checkpoint files should be stored, how certain files should be accessed, etc).

condor_collector This daemon is responsible for collecting all the information about the status of a Condor pool. All other daemons periodically send ClassAd updates to the collector. These ClassAds contain all the information about the state of the daemons, the resources they represent or resource requests in the pool (such as jobs that have been submitted to a given schedd). The *condor_status* command can be used to query the collector for specific information about various parts of Condor. In addition, the Condor daemons themselves query the collector for important information, such as what address to use for sending commands to a remote machine.

condor_negotiator This daemon is responsible for all the match-making within the Condor system. Periodically, the negotiator begins a *negotiation cycle*, where it queries the collector for the current state of all the resources in the pool. It contacts each schedd that has waiting resource requests in priority order, and tries to match available resources with those requests. The negotiator is responsible for enforcing user priorities in the system, where the more resources a given user has claimed, the less priority they have to acquire more resources. If a user with a better priority has jobs that are waiting to run, and resources are claimed by a user with a worse priority, the negotiator can preempt that resource and match it with the user with better priority.

NOTE: A higher numerical value of the user priority in Condor translate into worse priority for that user. The best priority you can have is 0.5, the lowest numerical value, and your priority gets worse as this number grows.

condor_kbdd This daemon is used on Linux and Windows. On those platforms, the *condor_startd* frequently cannot determine console (keyboard or mouse) activity directly from the system, and requires a separate process to do so. On Linux, the *condor_kbdd* connects to the X Server and periodically checks to see if there has been any activity. On Windows, the *condor_kbdd* runs as the logged-in user and registers with the system to receive keyboard and mouse events. When it detects console activity, the *condor_kbdd* sends a command to the startd. That way, the startd knows the machine owner is using the machine again and can perform whatever actions are necessary, given the policy it has been configured to enforce.

condor_ckpt_server This is the checkpoint server. It services requests to store and retrieve checkpoint files. If your pool is configured to use a checkpoint server but that machine (or the server itself is down) Condor will revert to sending the checkpoint files for a given job back to the submit machine.

condor_quill This daemon builds and manages a database that represents a copy of the Condor job queue. The *condor_q* and *condor_history* tools can then query the database.

condor_dbmsd This daemon assists the *condor_quill* daemon.

condor_gridmanager This daemon handles management and execution of all **grid** universe jobs. The *condor_schedd* invokes the *condor_gridmanager* when there are **grid** universe jobs in the queue, and the *condor_gridmanager* exits when there are no more **grid** universe jobs in the queue.

condor_credd This daemon runs on Windows platforms to manage password storage in a secure manner.

condor_had This daemon implements the high availability of a pool's central manager through monitoring the communication of necessary daemons. If the current, functioning, central manager machine stops working, then this daemon ensures that another machine takes its place, and becomes the central manager of the pool.

condor_replication This daemon assists the *condor_had* daemon by keeping an updated copy of the pool's state. This state provides a better transition from one machine to the next, in the event that the central manager machine stops working.

condor_transferer This short lived daemon is invoked by the *condor_replication* daemon to accomplish the task of transferring a state file before exiting.

condor_procd This daemon controls and monitors process families within Condor. Its use is optional in general but it must be used if privilege separation (see Section 3.6.14) or group-ID based tracking (see Section 3.13.12) is enabled.

condor_job_router This daemon transforms **vanilla** universe jobs into **grid** universe jobs, such that the transformed jobs are capable of running elsewhere, as appropriate.

condor_lease_manager This daemon manages leases in a persistent manner. Leases are represented by ClassAds.

condor_rooster This daemon wakes hibernating machines based upon configuration details.

condor_shared_port This daemon listens for incoming TCP packets on behalf of Condor daemons, thereby reducing the number of required ports that must be opened when Condor is accessible through a firewall.

condor_hdfs This daemon manages the configuration of a Hadoop file system as well as the invocation of a properly configured Hadoop file system.

See figure 3.1 for a graphical representation of the pool architecture.

3.2 Installation

This section contains the instructions for installing Condor. The installation will have a default configuration that can be customized. Sections of the manual that follow this one explain customization.

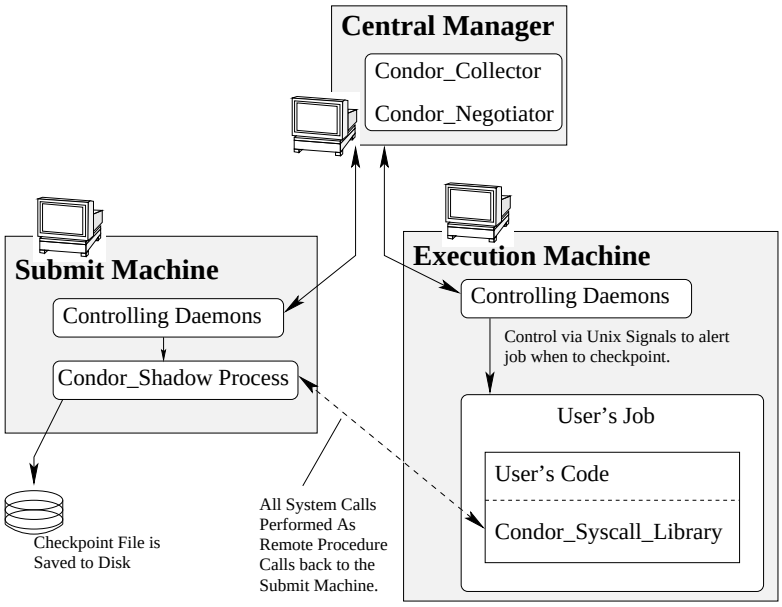

Figure 3.1: Pool Architecture

Read this entire section before starting installation.

Please read the copyright and disclaimer information in section on page xiv of the manual. Installation and use of Condor is acknowledgment that you have read and agree to the terms.

3.2.1 Obtaining Condor

The first step to installing Condor is to download it from the Condor web site, http://www.cs.wisc.edu/condor. The downloads are available from the downloads page, at http://www.cs.wisc.edu/condor/downloads/.

The platform-dependent Condor files are currently available from two sites. The main site is at the University of Wisconsin–Madison, Madison, Wisconsin, USA. A second site is the Istituto Nazionale di Fisica Nucleare Sezione di Bologna, Bologna, Italy. Please choose the site nearest to you.

Make note of the location of where you download the binary into.

The Condor binary distribution is packaged in the following files and directories:

DOC directions on where to find Condor documentation

INSTALL these installation directions

LICENSE.TXT the licensing agreement. By installing Condor, you agree to the contents of this file

README general information

condor_install the Perl script used to install and configure Condor

examples directory containing C, Fortran and C++ example programs to run with Condor

bin directory which contains the distribution Condor user programs.

sbin directory which contains the distribution Condor system programs.

etc directory which contains the distribution Condor configuration data.

lib directory which contains the distribution Condor libraries.

libexec directory which contains the distribution Condor programs that are only used internally by Condor.

man directory which contains the distribution Condor manual pages.

sql directory which contains the distribution Condor files used for SQL operations.

src directory which contains the distribution Condor source code for CHIRP and DRMAA.

Before you install, please consider joining the condor-world mailing list. Traffic on this list is kept to an absolute minimum. It is only used to announce new releases of Condor. To subscribe, send a message to majordomo@cs.wisc.edu with the body:

```
subscribe condor-world
```

3.2.2 Preparation

Before installation, make a few important decisions about the basic layout of your pool. The decisions answer the questions:

1. What machine will be the central manager?
2. What machines should be allowed to submit jobs?
3. Will Condor run as root or not?
4. Who will be administering Condor on the machines in your pool?
5. Will you have a Unix user named condor and will its home directory be shared?
6. Where should the machine-specific directories for Condor go?
7. Where should the parts of the Condor system be installed?

- Configuration files
- Release directory
  - user binaries
  - system binaries
  - `lib` directory
  - `etc` directory
- Documentation

8. Am I using AFS?
9. Do I have enough disk space for Condor?

1. What machine will be the central manager? One machine in your pool must be the central manager. Install Condor on this machine first. This is the centralized information repository for the Condor pool, and it is also the machine that does match-making between available machines and submitted jobs. If the central manager machine crashes, any currently active matches in the system will keep running, but no new matches will be made. Moreover, most Condor tools will stop working. Because of the importance of this machine for the proper functioning of Condor, install the central manager on a machine that is likely to stay up all the time, or on one that will be rebooted quickly if it does crash.

Also consider network traffic and your network layout when choosing your central manager. All the daemons send updates (by default, every 5 minutes) to this machine. Memory requirements for the central manager differ by the number of machines in the pool. A pool with up to about 100 machines will require approximately 25 Mbytes of memory for the central manager's tasks. A pool with about 1000 machines will require approximately 100 Mbytes of memory for the central manager's tasks.

A faster CPU will improve the time to do matchmaking.

2. Which machines should be allowed to submit jobs? Condor can restrict the machines allowed to submit jobs. Alternatively, it can allow any machine the network allows to connect to a submit machine to submit jobs. If the Condor pool is behind a firewall, and all machines inside the firewall are trusted, the `HOSTALLOW_WRITE` configuration entry can be set to *. Otherwise, it should be set to reflect the set of machines permitted to submit jobs to this pool. Condor tries to be secure by default, so out of the box, the configuration file ships with an invalid definition for this configuration variable. This invalid value allows no machine to connect and submit jobs, so after installation, change this entry. Look for the entry defined with the value `YOU_MUST_CHANGE_THIS_INVALID_CONDOR_CONFIGURATION_VALUE`.

3. Will Condor run as root or not? Start up the Condor daemons as the Unix user root. Without this, Condor can do very little to enforce security and policy decisions. You can install Condor as any user, however there are both serious security and performance consequences. Please see section 3.6.13 on page 350 in the manual for the details and ramifications of running Condor as a Unix user other than root.

4. **Who will administer Condor?** Either root will be administering Condor directly, or someone else would be acting as the Condor administrator. If root has delegated the responsibility to another person, keep in mind that as long as Condor is started up as root, it should be clearly understood that whoever has the ability to edit the condor configuration files can effectively run arbitrary programs as root.

5. **Will you have a Unix user named condor, and will its home directory be shared?** To simplify installation of Condor, create a Unix user named condor on all machines in the pool. The Condor daemons will create files (such as the log files) owned by this user, and the home directory can be used to specify the location of files and directories needed by Condor. The home directory of this user can either be shared among all machines in your pool, or could be a separate home directory on the local partition of each machine. Both approaches have advantages and disadvantages. Having the directories centralized can make administration easier, but also concentrates the resource usage such that you potentially need a lot of space for a single shared home directory. See the section below on machine-specific directories for more details.

   Note that the user condor must not be an account into which a person can log in. If a person can log in as user condor, it permits a major security breach, in that the user condor could submit jobs that run as any other user, providing complete access to the user's data by the jobs. A standard way of not allowing log in to an account on Unix platforms is to enter an invalid shell in the password file.

   If you choose not to create a user named condor, then you must specify either via the `CONDOR_IDS` environment variable or the `CONDOR_IDS` config file setting which uid.gid pair should be used for the ownership of various Condor files. See section 3.6.13 on UIDs in Condor on page 349 in the Administrator's Manual for details.

6. **Where should the machine-specific directories for Condor go?** Condor needs a few directories that are unique on every machine in your pool. These are `spool`, `log`, and `execute`. Generally, all three are subdirectories of a single machine specific directory called the local directory (specified by the `LOCAL_DIR` macro in the configuration file). Each should be owned by the user that Condor is to be run as.

   If you have a Unix user named condor with a local home directory on each machine, the `LOCAL_DIR` could just be user condor's home directory (`LOCAL_DIR` = `$(TILDE)` in the configuration file). If this user's home directory is shared among all machines in your pool, you would want to create a directory for each host (named by host name) for the local directory (for example, `LOCAL_DIR` = `$(TILDE)`/hosts/`$(HOSTNAME)`). If you do not have a condor account on your machines, you can put these directories wherever you'd like. However, where to place them will require some thought, as each one has its own resource needs:

   `execute` This is the directory that acts as the current working directory for any Condor jobs that run on a given execute machine. The binary for the remote job is copied into this directory, so there must be enough space for it. (Condor will not send a job to a machine that does not have enough disk space to hold the initial binary). In addition, if the remote job dumps core for some reason, it is first dumped to the execute directory before it is sent back to the submit machine. So, put the execute directory on a partition with enough space to hold a possible core file from the jobs submitted to your pool.

spool The `spool` directory holds the job queue and history files, and the checkpoint files for all jobs submitted from a given machine. As a result, disk space requirements for the `spool` directory can be quite large, particularly if users are submitting jobs with very large executables or image sizes. By using a checkpoint server (see section 3.8 on Installing a Checkpoint Server on page 384 for details), you can ease the disk space requirements, since all checkpoint files are stored on the server instead of the spool directories for each machine. However, the initial checkpoint files (the executables for all the clusters you submit) are still stored in the spool directory, so you will need some space, even with a checkpoint server.

log Each Condor daemon writes its own log file, and each log file is placed in the `log` directory. You can specify what size you want these files to grow to before they are rotated, so the disk space requirements of the directory are configurable. The larger the log files, the more historical information they will hold if there is a problem, but the more disk space they use up. If you have a network file system installed at your pool, you might want to place the log directories in a shared location (such as `/usr/local/condor/logs/$(HOSTNAME)`), so that you can view the log files from all your machines in a single location. However, if you take this approach, you will have to specify a local partition for the `lock` directory (see below).

lock Condor uses a small number of lock files to synchronize access to certain files that are shared between multiple daemons. Because of problems encountered with file locking and network file systems (particularly NFS), these lock files should be placed on a local partition on each machine. By default, they are placed in the `log` directory. If you place your `log` directory on a network file system partition, specify a local partition for the lock files with the `LOCK` parameter in the configuration file (such as `/var/lock/condor`).

Generally speaking, it is recommended that you do not put these directories (except `lock`) on the same partition as `/var`, since if the partition fills up, you will fill up `/var` as well. This will cause lots of problems for your machines. Ideally, you will have a separate partition for the Condor directories. Then, the only consequence of filling up the directories will be Condor's malfunction, not your whole machine.

7. Where should the parts of the Condor system be installed?

- Release directory
  - User Binaries
  - System Binaries
  - `lib` Directory
  - `etc` Directory
- Documentation
- Configuration Files

Configuration Files There are a number of configuration files that allow you different levels of control over how Condor is configured at each machine in your pool. The global configuration file is shared by all machines in the pool. For ease of administration, this file should be located on a shared file system, if possible. In addition, there is a local configuration file for each machine, where you can override settings in the global file.

This allows you to have different daemons running, different policies for when to start and stop Condor jobs, and so on. You can also have configuration files specific to each platform in your pool. See section 3.13.4 on page 435 about Configuring Condor for Multiple Platforms for details.

In general, there are a number of places that Condor will look to find its configuration files. The first file it looks for is the global configuration file. These locations are searched in order until a configuration file is found. If none contain a valid configuration file, Condor will print an error message and exit:

1. File specified in the `CONDOR_CONFIG` environment variable
2. `/etc/condor/condor_config`
3. `/usr/local/etc/condor_config`
4. `~condor/condor_config`
5. `$(GLOBUS_LOCATION)/etc/condor_config`

If you specify a file in the `CONDOR_CONFIG` environment variable and there's a problem reading that file, Condor will print an error message and exit right away, instead of continuing to search the other options. However, if no `CONDOR_CONFIG` environment variable is set, Condor will search through the other options.

Next, Condor tries to load the local configuration file(s). The only way to specify the local configuration file(s) is in the global configuration file, with the `LOCAL_CONFIG_FILE` macro. If that macro is not set, no local configuration file is used. This macro can be a list of files or a single file.

Release Directory Every binary distribution contains a contains five subdirectories: `bin`, `etc`, `lib`, `sbin`, and `libexec`. Wherever you choose to install these five directories we call the release directory (specified by the `RELEASE_DIR` macro in the configuration file). Each release directory contains platform-dependent binaries and libraries, so you will need to install a separate one for each kind of machine in your pool. For ease of administration, these directories should be located on a shared file system, if possible.

- User Binaries:
  All of the files in the `bin` directory are programs the end Condor users should expect to have in their path. You could either put them in a well known location (such as `/usr/local/condor/bin`) which you have Condor users add to their `PATH` environment variable, or copy those files directly into a well known place already in the user's PATHs (such as `/usr/local/bin`). With the above examples, you could also leave the binaries in `/usr/local/condor/bin` and put in soft links from `/usr/local/bin` to point to each program.
- System Binaries:
  All of the files in the `sbin` directory are Condor daemons and agents, or programs that only the Condor administrator would need to run. Therefore, add these programs only to the `PATH` of the Condor administrator.
- Private Condor Binaries:
  All of the files in the `libexec` directory are Condor programs that should never be run by hand, but are only used internally by Condor.
- `lib` Directory:

The files in the `lib` directory are the Condor libraries that must be linked in with user jobs for all of Condor's checkpointing and migration features to be used. `lib` also contains scripts used by the *condor_compile* program to help re-link jobs with the Condor libraries. These files should be placed in a location that is world-readable, but they do not need to be placed in anyone's `PATH`. The *condor_compile* script checks the configuration file for the location of the `lib` directory.

- `etc` Directory:
  `etc` contains an `examples` subdirectory which holds various example configuration files and other files used for installing Condor. `etc` is the recommended location to keep the master copy of your configuration files. You can put in soft links from one of the places mentioned above that Condor checks automatically to find its global configuration file.

Documentation The documentation provided with Condor is currently available in HTML, Postscript and PDF (Adobe Acrobat). It can be locally installed wherever is customary at your site. You can also find the Condor documentation on the web at: http://www.cs.wisc.edu/condor/manual.

7. Am I using AFS? If you are using AFS at your site, be sure to read the section 3.13.1 on page 431 in the manual. Condor does not currently have a way to authenticate itself to AFS. A solution is not ready for Version 7.6.0. This implies that you are probably not going to want to have the `LOCAL_DIR` for Condor on AFS. However, you can (and probably should) have the Condor `RELEASE_DIR` on AFS, so that you can share one copy of those files and upgrade them in a centralized location. You will also have to do something special if you submit jobs to Condor from a directory on AFS. Again, read manual section 3.13.1 for all the details.

8. Do I have enough disk space for Condor? Condor takes up a fair amount of space. This is another reason why it is a good idea to have it on a shared file system. The compressed downloads currently range from a low of about 100 Mbytes for Windows to about 500 Mbytes for Linux. The compressed source code takes approximately 16 Mbytes.

In addition, you will need a lot of disk space in the local directory of any machines that are submitting jobs to Condor. See question 6 above for details on this.

3.2.3 Newer Unix Installation Procedure

The Perl script *condor_configure* installs Condor. Command-line arguments specify all needed information to this script. The script can be executed multiple times, to modify or further set the configuration. *condor_configure* has been tested using Perl 5.003. Use this or a more recent version of Perl.

After download, all the files are in a compressed, tar format. They need to be untarred, as

```
tar xzf completename.tar.gz
```

After untarring, the directory will have the Perl scripts *condor_configure* and *condor_install*, as well

as a “bin”, “etc”, “examples”, “include”, “lib”, “libexec”, “man”, “sbin”, “sql” and “src” subdirectories.

condor_configure and *condor_install* are the same program, but have different default behaviors. *condor_install* is identical to running “*condor_configure* –install=.”. *condor_configure* and *condor_install* work on above directories (“sbin”, etc.). As the names imply, *condor_install* is used to install Condor, whereas *condor_configure* is used to modify the configuration of an existing Condor install.

condor_configure and *condor_install* are completely command-line driven; it is not interactive. Several command-line arguments are always needed with *condor_configure* and *condor_install*. The argument

```
--install=/path/to/release.
```

specifies the path to the Condor release directories (see above). The default for *condor_install* is “–install=.”. The argument

```
--install-dir=directory
```

or

```
--prefix=directory
```

specifies the path to the install directory.

The argument

```
--local-dir=directory
```

specifies the path to the local directory.

The **–type** option to *condor_configure* specifies one or more of the roles that a machine may take on within the Condor pool: central manager, submit or execute. These options are given in a comma separated list. So, if a machine is both a submit and execute machine, the proper command-line option is

```
--type=manager,execute
```

Install Condor on the central manager machine first. If Condor will run as root in this pool (Item 3 above), run *condor_install* as root, and it will install and set the file permissions correctly. On the central manager machine, run *condor_install* as follows.

```
% condor_install --prefix=~condor \
--local-dir=/scratch/condor --type=manager
```

To update the above Condor installation, for example, to also be submit machine:

```
% condor_configure --prefix=~condor \
--local-dir=/scratch/condor --type=manager,submit
```

As in the above example, the central manager can also be a submit point or and execute machine, but this is only recommended for very small pools. If this is the case, the **–type** option changes to `manager,execute` or `manager,submit` or `manager,submit,execute`.

After the central manager is installed, the execute and submit machines should then be configured. Decisions about whether to run Condor as root should be consistent throughout the pool. For each machine in the pool, run

```
% condor_install --prefix=~condor \
--local-dir=/scratch/condor --type=execute,submit
```

See the *condor_configure* manual page in section 9 on page 722 for details.

3.2.4 Starting Condor Under Unix After Installation

Now that Condor has been installed on the machine(s), there are a few things to check before starting up Condor.

1. Read through the <`release_dir`>`/etc/condor_config` file. There are a lot of possible settings and you should at least take a look at the first two main sections to make sure everything looks okay. In particular, you might want to set up security for Condor. See the section 3.6.1 on page 315 to learn how to do this.

2. For Linux platforms, run the *condor_kbdd* to monitor keyboard and mouse activity on all machines within the pool that will run a *condor_startd*; these are machines that execute jobs. To do this, the subsystem `KBDD` will need to be added to the `DAEMON_LIST` configuration variable definition.

   For Unix platforms other than Linux, Condor can monitor the activity of your mouse and keyboard, provided that you tell it where to look. You do this with the `CONSOLE_DEVICES` entry in the condor_startd section of the configuration file. On most platforms, reasonable defaults are provided. For example, the default device for the mouse is 'mouse', since most installations have a soft link from `/dev/mouse` that points to the right device (such as `tty00` if you have a serial mouse, `psaux` if you have a PS/2 bus mouse, etc). If you do not have a `/dev/mouse` link, you should either create one (you will be glad you did), or change the `CONSOLE_DEVICES` entry in Condor's configuration file. This entry is a comma separated list, so you can have any devices in `/dev` count as 'console devices' and activity will be reported in the condor_startd's ClassAd as `ConsoleIdleTime`.

3. (Linux only) Condor needs to be able to find the `utmp` file. According to the Linux File System Standard, this file should be `/var/run/utmp`. If Condor cannot find it there, it looks in `/var/adm/utmp`. If it still cannot find it, it gives up. So, if your Linux distribution places this file somewhere else, be sure to put a soft link from `/var/run/utmp` to point to the real location.

To start up the Condor daemons, execute <`release_dir`>`/sbin/condor_master`. This is the Condor master, whose only job in life is to make sure the other Condor daemons are running. The master keeps track of the daemons, restarts them if they crash, and periodically checks to see if you have installed new binaries (and if so, restarts the affected daemons).

If you are setting up your own pool, you should start Condor on your central manager machine first. If you have done a submit-only installation and are adding machines to an existing pool, the start order does not matter.

To ensure that Condor is running, you can run either:

```
ps -ef | egrep condor_
```

or

```
ps -aux | egrep condor_
```

depending on your flavor of Unix. On a central manager machine that can submit jobs as well as execute them, there will be processes for:

- condor_master
- condor_collector
- condor_negotiator
- condor_startd
- condor_schedd

On a central manager machine that does not submit jobs nor execute them, there will be processes for:

- condor_master
- condor_collector
- condor_negotiator

For a machine that only submits jobs, there will be processes for:

- condor_master
- condor_schedd

For a machine that only executes jobs, there will be processes for:

- condor_master
- condor_startd

Once you are sure the Condor daemons are running, check to make sure that they are communicating with each other. You can run *condor_status* to get a one line summary of the status of each machine in your pool.

Once you are sure Condor is working properly, you should add *condor_master* into your startup/bootup scripts (i.e. `/etc/rc`) so that your machine runs *condor_master* upon bootup. *condor_master* will then fire up the necessary Condor daemons whenever your machine is rebooted.

If your system uses System-V style init scripts, you can look in `<release_dir>/etc/examples/condor.boot` for a script that can be used to start and stop Condor automatically by init. Normally, you would install this script as `/etc/init.d/condor` and put in soft link from various directories (for example, `/etc/rc2.d`) that point back to `/etc/init.d/condor`. The exact location of these scripts and links will vary on different platforms.

If your system uses BSD style boot scripts, you probably have an `/etc/rc.local` file. Add a line to start up `<release_dir>/sbin/condor_master`.

Now that the Condor daemons are running, there are a few things you can and should do:

1. (Optional) Do a full install for the *condor_compile* script. condor_compile assists in linking jobs with the Condor libraries to take advantage of all of Condor's features. As it is currently installed, it will work by placing it in front of any of the following commands that you would normally use to link your code: gcc, g++, g77, cc, acc, c89, CC, f77, fort77 and ld. If you complete the full install, you will be able to use condor_compile with any command whatsoever, in particular, make. See section 3.13.5 on page 438 in the manual for directions.
2. Try building and submitting some test jobs. See `examples/README` for details.
3. If your site uses the AFS network file system, see section 3.13.1 on page 431 in the manual.
4. We strongly recommend that you start up Condor (run the *condor_master* daemon) as user root. If you must start Condor as some user other than root, see section 3.6.13 on page 350.

3.2.5 Installation on Windows

This section contains the instructions for installing the Windows version of Condor. The install program will set up a slightly customized configuration file that may be further customized after the

installation has completed.

Please read the copyright and disclaimer information in section on page xiv of the manual. Installation and use of Condor is acknowledgment that you have read and agree to the terms.

Be sure that the Condor tools are of the same version as the daemons installed. The Condor executable for distribution is packaged in a single file named similar to:

```
condor-7.4.3-winnt50-x86.msi
```

This file is approximately 80 Mbytes in size, and it may be removed once Condor is fully installed.

Before installing Condor, please consider joining the condor-world mailing list. Traffic on this list is kept to an absolute minimum. It is only used to announce new releases of Condor. To subscribe, follow the directions given at http://www.cs.wisc.edu/condor/mail-lists/.

For any installation, Condor services are installed and run as the Local System account. Running the Condor services as any other account (such as a domain user) is not supported and could be problematic.

Installation Requirements

- Condor for Windows requires Windows 2000 SP4, Windows XP SP2, or a more recent version.
- 300 megabytes of free disk space is recommended. Significantly more disk space could be desired to be able to run jobs with large data files.
- Condor for Windows will operate on either an NTFS or FAT file system. However, for security purposes, NTFS is preferred.
- Condor for Windows requires the Visual C++ 2008 C runtime library.

Preparing to Install Condor under Windows

Before installing the Windows version of Condor, there are two major decisions to make about the basic layout of the pool.

1. What machine will be the central manager?
2. Is there enough disk space for Condor?

If the answers to these questions are already known, skip to the Windows Installation Procedure section below, section 3.2.5 on page 139. If unsure, read on.

- What machine will be the central manager?

  One machine in your pool must be the central manager. This is the centralized information repository for the Condor pool and is also the machine that matches available machines with waiting jobs. If the central manager machine crashes, any currently active matches in the system will keep running, but no new matches will be made. Moreover, most Condor tools will stop working. Because of the importance of this machine for the proper functioning of Condor, we recommend installing it on a machine that is likely to stay up all the time, or at the very least, one that will be rebooted quickly if it does crash. Also, because all the services will send updates (by default every 5 minutes) to this machine, it is advisable to consider network traffic and network layout when choosing the central manager.

  Install Condor on the central manager before installing on the other machines within the pool.

- Is there enough disk space for Condor?

  The Condor release directory takes up a fair amount of space. The size requirement for the release directory is approximately 250 Mbytes. Condor itself, however, needs space to store all of the jobs and their input files. If there will be large numbers of jobs, consider installing Condor on a volume with a large amount of free space.

Installation Procedure Using the MSI Program

Installation of Condor must be done by a user with administrator privileges. After installation, the Condor services will be run under the local system account. When Condor is running a user job, however, it will run that user job with normal user permissions.

Download Condor, and start the installation process by running the installer. The Condor installation is completed by answering questions and choosing options within the following steps.

If Condor is already installed. If Condor has been previously installed, a dialog box will appear before the installation of Condor proceeds. The question asks if you wish to preserve your current Condor configuration files. Answer yes or no, as appropriate.

If you answer yes, your configuration files will not be changed, and you will proceed to the point where the new binaries will be installed.

If you answer no, then there will be a second question that asks if you want to use answers given during the previous installation as default answers.

STEP 1: License Agreement. The first step in installing Condor is a welcome screen and license agreement. You are reminded that it is best to run the installation when no other Windows programs are running. If you need to close other Windows programs, it is safe to cancel the installation and close them. You are asked to agree to the license. Answer yes or no. If you should disagree with the License, the installation will not continue.

Also fill in name and company information, or use the defaults as given.

STEP 2: Condor Pool Configuration. The Condor configuration needs to be set based upon if this is a new pool or to join an existing one. Choose the appropriate radio button.

For a new pool, enter a chosen name for the pool. To join an existing pool, enter the host name of the central manager of the pool.

STEP 3: This Machine's Roles. Each machine within a Condor pool may either submit jobs or execute submitted jobs, or both submit and execute jobs. A check box determines if this machine will be a submit point for the pool.

A set of radio buttons determines the ability and configuration of the ability to execute jobs. There are four choices:

Do not run jobs on this machine. This machine will not execute Condor jobs.

Always run jobs and never suspend them.

Run jobs when the keyboard has been idle for 15 minutes.

Run jobs when the keyboard has been idle for 15 minutes, and the CPU is idle.

For testing purposes, it is often helpful to use the always run Condor jobs option.

For a machine that is to execute jobs and the choice is one of the last two in the list, Condor needs to further know what to do with the currently running jobs. There are two choices:

Keep the job in memory and continue when the machine meets the condition chosen for when to run jobs.

Restart the job on a different machine.

This choice involves a trade off. Restarting the job on a different machine is less intrusive on the workstation owner than leaving the job in memory for a later time. A suspended job left in memory will require swap space, which could be a scarce resource. Leaving a job in memory, however, has the benefit that accumulated run time is not lost for a partially completed job.

STEP 4: The Account Domain. Enter the machine's accounting (or UID) domain. On this version of Condor for Windows, this setting is only used for user priorities (see section 3.4 on page 275) and to form a default e-mail address for the user.

STEP 5: E-mail Settings. Various parts of Condor will send e-mail to a Condor administrator if something goes wrong and requires human attention. Specify the e-mail address and the SMTP relay host of this administrator. Please pay close attention to this e-mail, since it will indicate problems in the Condor pool.

STEP 6: Java Settings. In order to run jobs in the **java** universe, Condor must have the path to the jvm executable on the machine. The installer will search for and list the jvm path, if it finds one. If not, enter the path. To disable use of the **java** universe, leave the field blank.

STEP 7: Host Permission Settings. Machines within the Condor pool will need various types of access permission. The three categories of permission are read, write, and administrator. Enter the machines or domain to be given access permissions, or use the defaults provided. Wild cards and macros are permitted.

Read Read access allows a machine to obtain information about Condor such as the status of machines in the pool and the job queues. All machines in the pool should be given read access. In addition, giving read access to *.cs.wisc.edu will allow the Condor team to obtain information about the Condor pool, in the event that debugging is needed.

Write All machines in the pool should be given write access. It allows the machines you specify to send information to your local Condor daemons, for example, to start a Condor job. Note that for a machine to join the Condor pool, it must have both read and write access to all of the machines in the pool.

Administrator A machine with administrator access will be allowed more extended permission to do things such as change other user's priorities, modify the job queue, turn Condor services on and off, and restart Condor. The central manager should be given administrator access and is the default listed. This setting is granted to the entire machine, so care should be taken not to make this too open.

For more details on these access permissions, and others that can be manually changed in your configuration file, please see the section titled Setting Up IP/Host-Based Security in Condor in section section 3.6.9 on page 342.

STEP 8: VM Universe Setting. A radio button determines whether this machine will be configured to run **vm** universe jobs utilizing VMware. In addition to having the VMware Server installed, Condor also needs *Perl* installed. The resources available for **vm** universe jobs can be tuned with these settings, or the defaults listed may be used.

Version Use the default value, as only one version is currently supported.

Maximum Memory The maximum memory that each virtual machine is permitted to use on the target machine.

Maximum Number of VMs The number of virtual machines that can be run in parallel on the target machine.

Networking Support The VMware instances can be configured to use network support. There are four options in the pull-down menu.

- None: No networking support.
- NAT: Network address translation.
- Bridged: Bridged mode.
- NAT and Bridged: Allow both methods.

Path to Perl Executable The path to the *Perl* executable.

STEP 9: HDFS Settings. A radio button enables support for the Hadoop Distributed File System (HDFS). When enabled, a further radio button specifies either name node or data node mode.

Running HDFS requires Java to be installed, and Condor must know where the installation is. Running HDFS in data node mode also requires the installation of Cygwin, and the path to the Cygwin directory must be added to the global PATH environment variable.

HDFS has several configuration options that must be filled in to be used.

Primary Name Node The full host name of the primary name node.

Name Node Port The port that the name node is listening on.

Name Node Web Port The port the name node's web interface is bound to. It should be different from the name node's main port.

STEP 10: Choose Setup Type The next step is where the destination of the Condor files will be decided. We recommend that Condor be installed in the location shown as the default in the install choice: `C:\Condor`. This is due to several hard coded paths in scripts and configuration files. Clicking on the Custom choice permits changing the installation directory.

Installation on the local disk is chosen for several reasons. The Condor services run as local system, and within Microsoft Windows, local system has no network privileges. Therefore, for Condor to operate, Condor should be installed on a local hard drive, as opposed to a network drive (file server).

The second reason for installation on the local disk is that the Windows usage of drive letters has implications for where Condor is placed. The drive letter used must be not change, even when different users are logged in. Local drive letters do not change under normal operation of Windows.

While it is strongly discouraged, it may be possible to place Condor on a hard drive that is not local, if a dependency is added to the service control manager such that Condor starts after the required file services are available.

Unattended Installation Procedure Using the Included Set Up Program

This section details how to run the Condor for Windows installer in an unattended batch mode. This mode is one that occurs completely from the command prompt, without the GUI interface.

The Condor for Windows installer uses the Microsoft Installer (MSI) technology, and it can be configured for unattended installs analogous to any other ordinary MSI installer.

The following is a sample batch file that is used to set all the properties necessary for an unattended install.

```
@echo on
set ARGS=
set ARGS=NEWPOOL="N"
set ARGS=%ARGS% POOLNAME=""
set ARGS=%ARGS% RUNJOBS="C"
set ARGS=%ARGS% VACATEJOBS="Y"
set ARGS=%ARGS% SUBMITJOBS="Y"
set ARGS=%ARGS% CONDOREMAIL="you@yours.com"
set ARGS=%ARGS% SMTPSERVER="smtp.localhost"
set ARGS=%ARGS% HOSTALLOWREAD="*"
set ARGS=%ARGS% HOSTALLOWWRITE="*"
set ARGS=%ARGS% HOSTALLOWADMINISTRATOR="$(IP_ADDRESS)"
set ARGS=%ARGS% INSTALLDIR="C:\Condor"
set ARGS=%ARGS% POOLHOSTNAME="$(IP_ADDRESS)"
set ARGS=%ARGS% ACCOUNTINGDOMAIN="none"
set ARGS=%ARGS% JVMLOCATION="C:\Windows\system32\java.exe"
set ARGS=%ARGS% USEVMUNIVERSE="N"set ARGS=%ARGS% VMMEMORY="128"
```

```
set ARGS=%ARGS% VMMAXNUMBER="$(NUM_CPUS)"
set ARGS=%ARGS% VMNETWORKING="N"
set ARGS=%ARGS% USEHDFS="N"
set ARGS=%ARGS% NAMENODE=""
set ARGS=%ARGS% HDFSMODE="HDFS_NAMENODE"
set ARGS=%ARGS% HDFSPORT="5000"
set ARGS=%ARGS% HDFSWEBPORT="4000"

msiexec /qb /l* condor-install-log.txt /i condor-7.1.0-winnt50-x86.msi %ARGS%
```

Each property corresponds to answers that would have been supplied while running an interactive installer. The following is a brief explanation of each property as it applies to unattended installations:

NEWPOOL = < **Y** | **N** > determines whether the installer will create a new pool with the target machine as the central manager.

POOLNAME sets the name of the pool, if a new pool is to be created. Possible values are either the name or the empty string `""`.

RUNJOBS = < **N** | **A** | **I** | **C** > determines when Condor will run jobs. This can be set to:

- Never run jobs (N)
- Always run jobs (A)
- Only run jobs when the keyboard and mouse are Idle (I)
- Only run jobs when the keyboard and mouse are idle and the CPU usage is low (C)

VACATEJOBS = < **Y** | **N** > determines what Condor should do when it has to stop the execution of a user job. When set to Y, Condor will vacate the job and start it somewhere else if possible. When set to N, Condor will merely suspend the job in memory and wait for the machine to become available again.

SUBMITJOBS = < **Y** | **N** > will cause the installer to configure the machine as a submit node when set to Y.

CONDOREMAIL sets the e-mail address of the Condor administrator. Possible values are an e-mail address or the empty string `""`.

HOSTALLOWREAD is a list of host names that are allowed to issue READ commands to Condor daemons. This value should be set in accordance with the `HOSTALLOW_READ` setting in the configuration file, as described in section 3.6.9 on page 342.

HOSTALLOWWRITE is a list of host names that are allowed to issue WRITE commands to Condor daemons. This value should be set in accordance with the `HOSTALLOW_WRITE` setting in the configuration file, as described in section 3.6.9 on page 342.

HOSTALLOWADMINISTRATOR is a list of host names that are allowed to issue ADMINISTRATOR commands to Condor daemons. This value should be set in accordance with the `HOSTALLOW_ADMINISTRATOR` setting in the configuration file, as described in section 3.6.9 on page 342.

INSTALLDIR defines the path to the directory where Condor will be installed.

POOLHOSTNAME defines the host name of the pool's central manager.

ACCOUNTINGDOMAIN defines the accounting (or UID) domain the target machine will be in.

JVMLOCATION defines the path to Java virtual machine on the target machine.

SMTPSERVER defines the host name of the SMTP server that the target machine is to use to send e-mail.

VMMEMORY an integer value that defines the maximum memory each VM run on the target machine.

VMMAXNUMBER an integer value that defines the number of VMs that can be run in parallel on the target machine.

VMNETWORKING = $<$ **N** $|$ **A** $|$ **B** $|$ **C** $>$ determines if VM Universe can use networking. This can be set to:

- None (N)
- NAT (A)
- Bridged (B)
- NAT and Bridged (C)

USEVMUNIVERSE = $<$ **Y** $|$ **N** $>$ will cause the installer to enable VM Universe jobs on the target machine.

PERLLOCATION defines the path to *Perl* on the target machine. This is required in order to use the **vm** universe.

USEHDFS = $<$ **Y** $|$ **N** $>$ determines if HDFS is run.

HDFSMODE $<$ **HDFS_DATANODE** $|$ **HDFS_NAMENODE** $>$ sets the mode HDFS runs in.

NAMENODE sets the host name of the primary name node.

HDFSPORT sets the port number that the primary name node listens to.

HDFSWEBPORT sets the port number that the name node web interface is bound to.

After defining each of these properties for the MSI installer, the installer can be started with the *msiexec* command. The following command starts the installer in unattended mode, and it dumps a journal of the installer's progress to a log file:

```
msiexec /qb /lxv* condor-install-log.txt /i condor-7.2.2-winnt50-x86.msi [property=value] ...
```

More information on the features of *msiexec* can be found at Microsoft's website at http://www.microsoft.com/resources/documentation/windows/xp/all/proddocs/en-us/msiexec.mspx.

Manual Installation Condor on Windows

If you are to install Condor on many different machines, you may wish to use some other mechanism to install Condor on additional machines rather than running the Setup program described above on each machine.

WARNING: This is for advanced users only! All others should use the Setup program described above.

Here is a brief overview of how to install Condor manually without using the provided GUI-based setup program:

The Service The service that Condor will install is called "Condor". The Startup Type is Automatic. The service should log on as System Account, but **do not enable** "Allow Service to Interact with Desktop". The program that is run is *condor_master.exe*.

The Condor service can be installed and removed using the `sc.exe` tool, which is included in Windows XP and Windows 2003 Server. The tool is also available as part of the Windows 2000 Resource Kit.

Installation can be done as follows:

```
sc create Condor binpath= c:\condor\bin\condor_master.exe
```

To remove the service, use:

```
sc delete Condor
```

The Registry Condor uses a few registry entries in its operation. The key that Condor uses is HKEY_LOCAL_MACHINE/Software/Condor. The values that Condor puts in this registry key serve two purposes.

1. The values of CONDOR_CONFIG and RELEASE_DIR are used for Condor to start its service.

   CONDOR_CONFIG should point to the `condor_config` file. In this version of Condor, it **must** reside on the local disk.

   RELEASE_DIR should point to the directory where Condor is installed. This is typically `C:\Condor`, and again, this **must** reside on the local disk.

2. The other purpose is storing the entries from the last installation so that they can be used for the next one.

The File System The files that are needed for Condor to operate are identical to the Unix version of Condor, except that executable files end in `.exe`. For example the on Unix one of the files is `condor_master` and on Condor the corresponding file is `condor_master.exe`.

These files currently must reside on the local disk for a variety of reasons. Advanced Windows users might be able to put the files on remote resources. The main concern is twofold. First, the files must be there when the service is started. Second, the files must always be in the same spot (including drive letter), no matter who is logged into the machine.

Note also that when installing manually, you will need to create the directories that Condor will expect to be present given your configuration. This normally is simply a matter of creating the `log`, `spool`, and `execute` directories.

Starting Condor Under Windows After Installation

After the installation of Condor is completed, the Condor service must be started. If you used the GUI-based setup program to install Condor, the Condor service should already be started. If you installed manually, Condor must be started by hand, or you can simply reboot. NOTE: The Condor service will start automatically whenever you reboot your machine.

To start Condor by hand:

1. From the Start menu, choose Settings.
2. From the Settings menu, choose Control Panel.
3. From the Control Panel, choose Services.
4. From Services, choose Condor, and Start.

Or, alternatively you can enter the following command from a command prompt:

```
net start condor
```

Run the Task Manager (Control-Shift-Escape) to check that Condor services are running. The following tasks should be running:

- *condor_master.exe*
- *condor_negotiator.exe*, if this machine is a central manager.
- *condor_collector.exe*, if this machine is a central manager.
- *condor_startd.exe*, if you indicated that this Condor node should start jobs
- *condor_schedd.exe*, if you indicated that this Condor node should submit jobs to the Condor pool.

Also, you should now be able to open up a new cmd (DOS prompt) window, and the Condor bin directory should be in your path, so you can issue the normal Condor commands, such as *condor_q* and *condor_status*.

Condor is Running Under Windows ... Now What?

Once Condor services are running, try submitting test jobs. Example 2 within section 2.5.1 on page 19 presents a vanilla universe job.

3.2.6 RPMs

RPMs are available in Condor Version 7.6.0. We provide a Yum repository, as well as installation and configuration in one easy step. This RPM installation is currently available for Red Hat-compatible systems only. As of Condor version 7.5.1, the Condor RPM installs into FHS locations.

Yum repositories are at http://www.cs.wisc.edu/condor/yum/ . The repositories are named to distinguish stable releases from development releases and by Red Hat version number. The 4 repositories are:

- `condor-stable-rhel4.repo`
- `condor-stable-rhel5.repo`
- `condor-development-rhel4.repo`
- `condor-development-rhel5.repo`

Here are an ordered set of steps that get Condor running using the RPM.

1. The Condor package will automatically add a `condor` user/group, if it does not exist already. Sites wishing to control the attributes of this user/group should add the `condor` user/group manually before installation.
2.
3. Download and install the meta-data that describes the appropriate YUM repository. This example is for the stable series, on RHEL 5.

   ```
   cd /etc/yum.repos.d
   wget http://www.cs.wisc.edu/condor/yum/repo.d/condor-stable-rhel5.repo
   ```

   Note that this step need be done only once; do not get the same repository more than once.
4. Install Condor. For 32-bit machines:

   ```
   yum install condor
   ```

   For 64-bit machines:

   ```
   yum install condor.x86_64
   ```
5. As needed, edit the Condor configuration files to customize. The configuration files are in the directory `/etc/condor/` . Do *not* use *condor_configure* or *condor_install* for configuration. The installation will be able to find configuration files without additional administrative intervention, as the configuration files are placed in `/etc`, and Condor searches this directory.
6. Start Condor daemons:

   ```
   /sbin/service condor start
   ```

3.2.7 Debian Packages

Debian packages are available in Condor Version 7.6.0. We provide an APT repository, as well as installation and configuration in one easy step. These Debian packages of Condor are currently available for Debian 4 (Etch) and Debian 5 (Lenny). As of Condor version 7.5.1, the Condor Debian package installs into FHS locations.

The Condor APT repositories are specified at http://www.cs.wisc.edu/condor/debian/ . See this web page for repository information.

Here are an ordered set of steps that get Condor running.

1. The Condor package will automatically add a `condor` user/group, if it does not exist already. Sites wishing to control the attributes of this user/group should add the `condor` user/group manually before installation.
2. If not already present, set up access to the appropriate APT repository; they are distinguished as stable or development release, and by operating system. Ensure that the correct one of the following release and operating system-specific lines is in the file `/etc/apt/sources.list` .

   ```
   deb http://www.cs.wisc.edu/condor/debian/stable/ etch contrib
   deb http://www.cs.wisc.edu/condor/debian/development/ etch contrib
   deb http://www.cs.wisc.edu/condor/debian/stable/ lenny contrib
   deb http://www.cs.wisc.edu/condor/debian/development/ lenny contrib
   ```

   Note that this step need be done only once; do not add the same repository more than once.
3. Install and start Condor services:

   ```
   apt-get update
   apt-get install condor
   ```
4. As needed, edit the Condor configuration files to customize. The configuration files are in the directory `/etc/condor/` . Do *not* use *condor_configure* or *condor_install* for configuration. The installation will be able to find configuration files without additional administrative intervention, as the configuration files are placed in `/etc`, and Condor searches this directory.

   Then, if any configuration changes are made, restart Condor with

   ```
   /etc/init.d/condor restart
   ```

3.2.8 Upgrading - Installing a Newer Version of Condor

Section 3.10.1 on page 392 within the section on Pool Management describes strategies for doing an upgrade: changing the running version of Condor from the current installation to a newer version.

3.2.9 Installing the CondorView Client Contrib Module

The CondorView Client contrib module is used to automatically generate World Wide Web pages to display usage statistics of a Condor pool. Included in the module is a shell script which invokes the *condor_stats* command to retrieve pool usage statistics from the CondorView server, and generate HTML pages from the results. Also included is a Java applet, which graphically visualizes Condor usage information. Users can interact with the applet to customize the visualization and to zoom in to a specific time frame. Figure 3.2 on page 149 is a screen shot of a web page created by CondorView. To get a further feel for what pages generated by CondorView look like, view the statistics for the University of Wisconsin-Madison pool by visiting the URL http://condor-view.cs.wisc.edu/condor-view-applet.

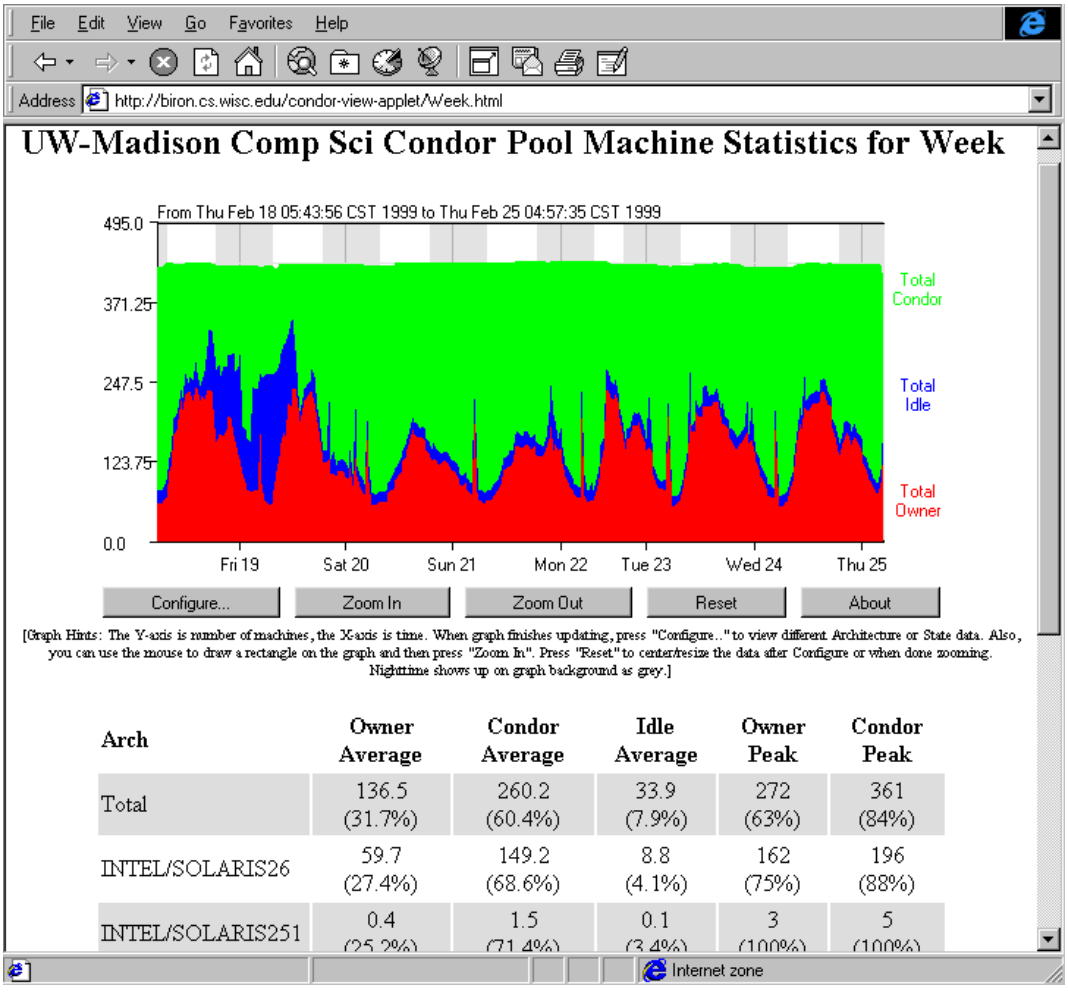

Figure 3.2: Screen shot of CondorView Client

After unpacking and installing the CondorView Client, a script named *make_stats* can be invoked to create HTML pages displaying Condor usage for the past hour, day, week, or month. By using the Unix *cron* facility to periodically execute *make_stats*, Condor pool usage statistics can be kept up to date automatically. This simple model allows the CondorView Client to be easily installed; no Web server CGI interface is needed.

Step-by-Step Installation of the CondorView Client

1. Make certain that the CondorView Server is configured. Section 3.13.7 describes configuration of the server. The server logs information on disk in order to provide a persistent, historical database of pool statistics. The CondorView Client makes queries over the network to this database. The *condor_collector* includes this database support. To activate the persistent database logging, add the following entries to the configuration file for the *condor_collector* chosen to act as the ViewServer.

   ```
   POOL_HISTORY_DIR = /full/path/to/directory/to/store/historical/data
   KEEP_POOL_HISTORY = True
   ```

2. Create a directory where CondorView is to place the HTML files. This directory should be one published by a web server, so that HTML files which exist in this directory can be accessed using a web browser. This directory is referred to as the `VIEWDIR` directory.

3. Download the *view_client* contrib module. Follow links for contrib modules on the downloads page at http://www.cs.wisc.edu/condor/downloads-v2/download.pl.

4. Unpack or untar this contrib module into the directory `VIEWDIR`. This creates several files and subdirectories. Further unpack the jar file within the `VIEWDIR` directory with:

   ```
   jar -xf condorview.jar
   ```

5. Edit the *make_stats* script. At the beginning of the file are six parameters to customize. The parameters are

   ORGNAME A brief name that identifies an organization. An example is "Univ of Wisconsin". Do not use any slashes in the name or other special regular-expression characters. Avoid the characters \ ^ and $.

   CONDORADMIN The e-mail address of the Condor administrator at your site. This e-mail address will appear at the bottom of the web pages.

   VIEWDIR The full path name (*not* a relative path) to the `VIEWDIR` directory set by installation step 2. It is the directory that contains the *make_stats* script.

   STATSDIR The full path name of the directory which contains the *condor_stats* binary. The *condor_stats* program is included in the <`release_dir`>`/bin` directory. The value for `STATSDIR` is added to the `PATH` parameter by default.

PATH A list of subdirectories, separated by colons, where the *make_stats* script can find the *awk*, *bc*, *sed*, *date*, and *condor_stats* programs. If *perl* is installed, the path should also include the directory where *perl* is installed. The following default works on most systems:

```
PATH=/bin:/usr/bin:$STATSDIR:/usr/local/bin
```

6. To create all of the initial HTML files, run

   ```
   ./make_stats setup
   ```

   Open the file `index.html` to verify that things look good.

7. Add the *make_stats* program to *cron*. Running *make_stats* in step 6 created a `cronentries` file. This `cronentries` file is ready to be processed by the Unix *crontab* command. The *crontab* manual page contains details about the *crontab* command and the *cron* daemon. Look at the `cronentries` file; by default, it will run *make_stats hour* every 15 minutes, *make_stats day* once an hour, *make_stats week* twice per day, and *make_stats month* once per day. These are reasonable defaults. Add these commands to cron on any system that can access the `VIEWDIR` and `STATSDIR` directories, even on a system that does not have Condor installed. The commands do not need to run as root user; in fact, they should probably not run as root. These commands can run as any user that has read/write access to the `VIEWDIR` directory. The command

   ```
   crontab cronentries
   ```

   can set the crontab file; note that this command overwrites the current, existing crontab file with the entries from the file `cronentries`.

8. Point the web browser at the `VIEWDIR` directory to complete the installation.

3.2.10 Dynamic Deployment

Dynamic deployment is a mechanism that allows rapid, automated installation and start up of Condor resources on a given machine. In this way any machine can be added to a Condor pool. The dynamic deployment tool set also provides tools to remove a machine from the pool, without leaving residual effects on the machine such as leftover installations, log files, and working directories.

Installation and start up is provided by *condor_cold_start*. The *condor_cold_start* program determines the operating system and architecture of the target machine, and transfers the correct installation package from an ftp, http, or grid ftp site. After transfer, it installs Condor and creates a local working directory for Condor to run in. As a last step, *condor_cold_start* begins running Condor in a manner which allows for later easy and reliable shut down.

The program that reliably shuts down and uninstalls a previously dynamically installed Condor instance is *condor_cold_stop*. *condor_cold_stop* begins by safely and reliably shutting off the running Condor installation. It ensures that Condor has completely shut down before continuing, and

optionally ensures that there are no queued jobs at the site. Next, *condor_cold_stop* removes and optionally archives the Condor working directories, including the `log` directory. These archives can be stored to a mounted file system or to a grid ftp site. As a last step, *condor_cold_stop* uninstalls the Condor executables and libraries. The end result is that the machine resources are left unchanged after a dynamic deployment of Condor leaves.

Configuration and Usage

Dynamic deployment is designed for the expert Condor user and administrator. Tool design choices were made for functionality, not ease-of-use.

Like every installation of Condor, a dynamically deployed installation relies on a configuration. To add a target machine to a previously created Condor pool, the global configuration file for that pool is a good starting point. Modifications to that configuration can be made in a separate, local configuration file used in the dynamic deployment. The global configuration file must be placed on an ftp, http, grid ftp, or file server accessible by *condor_cold_start*. The local configuration file is to be on a file system accessible by the target machine. There are some specific configuration variables that may be set for dynamic deployment. A list of executables and directories which must be present for Condor to start on the target machine may be set with the configuration variables `DEPLOYMENT_REQUIRED_EXECS` and `DEPLOYMENT_REQUIRED_DIRS`. If defined and the comma-separated list of executables or directories are not present, then *condor_cold_start* exits with error. Note this does not affect what is installed, only whether start up is successful.

A list of executables and directories which are recommended to be present for Condor to start on the target machine may be set with the configuration variables `DEPLOYMENT_RECOMMENDED_EXECS` and `DEPLOYMENT_RECOMMENDED_DIRS`. If defined and the comma-separated lists of executables or directories are not present, then *condor_cold_start* prints a warning message and continues. Here is a portion of the configuration relevant to a dynamic deployment of a Condor submit node:

```
DEPLOYMENT_REQUIRED_EXECS    = MASTER, SCHEDD, PREEN, STARTER, \
                               STARTER_STANDARD, SHADOW, \
                               SHADOW_STANDARD, GRIDMANAGER, GAHP, CONDOR_GAHP
DEPLOYMENT_REQUIRED_DIRS     = SPOOL, LOG, EXECUTE
DEPLOYMENT_RECOMMENDED_EXECS = CREDD
DEPLOYMENT_RECOMMENDED_DIRS  = LIB, LIBEXEC
```

Additionally, the user must specify which Condor services will be started. This is done through the `DAEMON_LIST` configuration variable. Another excerpt from a dynamic submit node deployment configuration:

```
DAEMON_LIST  = MASTER, SCHEDD
```

Finally, the location of the dynamically installed Condor executables is tricky to set, since the location is unknown before installation. Therefore, the variable `DEPLOYMENT_RELEASE_DIR` is

defined in the environment. It corresponds to the location of the dynamic Condor installation. If, as is often the case, the configuration file specifies the location of Condor executables in relation to the RELEASE_DIR variable, the configuration can be made dynamically deployable by setting RELEASE_DIR to DEPLOYMENT_RELEASE_DIR as

```
RELEASE_DIR = $(DEPLOYMENT_RELEASE_DIR)
```

In addition to setting up the configuration, the user must also determine where the installation package will reside. The installation package can be in either tar or gzipped tar form, and may reside on a ftp, http, grid ftp, or file server. Create this installation package by tar'ing up the binaries and libraries needed, and place them on the appropriate server. The binaries can be tar'ed in a flat structure or within bin and sbin. Here is a list of files to give an example structure for a dynamic deployment of the *condor_schedd* daemon.

```
% tar tfz latest-i686-Linux-2.4.21-37.ELsmp.tar.gz
bin/
bin/condor_config_val
bin/condor_q
sbin/
sbin/condor_preen
sbin/condor_shadow.std
sbin/condor_starter.std
sbin/condor_schedd
sbin/condor_master
sbin/condor_gridmanager
sbin/gt4_gahp
sbin/gahp_server
sbin/condor_starter
sbin/condor_shadow
sbin/condor_c-gahp
sbin/condor_off
```

3.3 Configuration

This section describes how to configure all parts of the Condor system. General information about the configuration files and their syntax is followed by a description of settings that affect all Condor daemons and tools. The settings that control the policy under which Condor will start, suspend, resume, vacate or kill jobs are described in section 3.5 on Startd Policy Configuration.

3.3.1 Introduction to Configuration Files

The Condor configuration files are used to customize how Condor operates at a given site. The basic configuration as shipped with Condor works well for most sites.

Each Condor program will, as part of its initialization process, configure itself by calling a library routine which parses the various configuration files that might be used including pool-wide,

platform-specific, and machine-specific configuration files. Environment variables may also contribute to the configuration.

The result of configuration is a list of key/value pairs. Each key is a configuration variable name, and each value is a string literal that may utilize macro substitution (as defined below). Some configuration variables are evaluated by Condor as ClassAd expressions; some are not. Consult the documentation for each specific case. Unless otherwise noted, configuration values that are expected to be numeric or boolean constants may be any valid ClassAd expression of operators on constants. Example:

```
MINUTE          = 60
HOUR            = (60 * $(MINUTE))
SHUTDOWN_GRACEFUL_TIMEOUT = ($(HOUR)*24)
```

Ordered Evaluation to Set the Configuration

Multiple files, as well as a program's environment variables determine the configuration. The order in which attributes are defined is important, as later definitions override existing definitions. The order in which the (multiple) configuration files are parsed is designed to ensure the security of the system. Attributes which must be set a specific way must appear in the last file to be parsed. This prevents both the naive and the malicious Condor user from subverting the system through its configuration. The order in which items are parsed is

1. global configuration file
2. local configuration file
3. specific environment variables prefixed with `_CONDOR_`

The locations for these files are as given in section 3.2.2 on page 131.

Some Condor tools utilize environment variables to set their configuration. These tools search for specifically-named environment variables. The variables are prefixed by the string `_CONDOR_` or `_condor_`. The tools strip off the prefix, and utilize what remains as configuration. As the use of environment variables is the last within the ordered evaluation, the environment variable definition is used. The security of the system is not compromised, as only specific variables are considered for definition in this manner, not any environment variables with the `_CONDOR_` prefix.

Configuration File Macros

Macro definitions are of the form:

```
<macro_name> = <macro_definition>
```

The macro name given on the left hand side of the definition is a case sensitive identifier. There must be white space between the macro name, the equals sign (=), and the macro definition. The macro definition is a string literal that may utilize macro substitution.

Macro invocations are of the form:

```
$(macro_name)
```

Macro definitions may contain references to other macros, even ones that are not yet defined, as long as they are eventually defined in the configuration files. All macro expansion is done after all configuration files have been parsed, with the exception of macros that reference themselves.

```
A = xxx
C = $(A)
```

is a legal set of macro definitions, and the resulting value of `C` is `xxx`. Note that `C` is actually bound to `$(A)`, not its value.

As a further example,

```
A = xxx
C = $(A)
A = yyy
```

is also a legal set of macro definitions, and the resulting value of `C` is `yyy`.

A macro may be incrementally defined by invoking itself in its definition. For example,

```
A = xxx
B = $(A)
A = $(A)yyy
A = $(A)zzz
```

is a legal set of macro definitions, and the resulting value of `A` is `xxxyyyzzz`. Note that invocations of a macro in its own definition are immediately expanded. `$(A)` is immediately expanded in line 3 of the example. If it were not, then the definition would be impossible to evaluate.

Recursively defined macros such as

```
A = $(B)
B = $(A)
```

are not allowed. They create definitions that Condor refuses to parse.

All entries in a configuration file must have an operator, which will be an equals sign (=). Identifiers are alphanumerics combined with the underscore character, optionally with a subsystem name

and a period as a prefix. As a special case, a line without an operator that begins with a left square bracket will be ignored. The following two-line example treats the first line as a comment, and correctly handles the second line.

```
[Condor Settings]
my_classad = [ foo=bar ]
```

To simplify pool administration, any configuration variable name may be prefixed by a subsystem (see the `$(SUBSYSTEM)` macro in section 3.3.1 for the list of subsystems) and the period (`.`) character. For configuration variables defined this way, the value is applied to the specific subsystem. For example, the ports that Condor may use can be restricted to a range using the `HIGHPORT` and `LOWPORT` configuration variables. If the range of intended ports is different for specific daemons, this syntax may be used.

```
MASTER.LOWPORT   = 20000
MASTER.HIGHPORT  = 20100
NEGOTIATOR.LOWPORT   =  22000
NEGOTIATOR.HIGHPORT  =  22100
```

Note that all configuration variables may utilize this syntax, but nonsense configuration variables may result. For example, it makes no sense to define

```
NEGOTIATOR.MASTER_UPDATE_INTERVAL = 60
```

since the *condor_negotiator* daemon does not use the `MASTER_UPDATE_INTERVAL` variable.

It makes little sense to do so, but Condor will configure correctly with a definition such as

```
MASTER.MASTER_UPDATE_INTERVAL = 60
```

The *condor_master* uses this configuration variable, and the prefix of `MASTER.` causes this configuration to be specific to the *condor_master* daemon.

This syntax has been further expanded to allow for the specification of a local name on the command line using the command line option

```
-local-name <local-name>
```

This allows multiple instances of a daemon to be run by the same *condor_master* daemon, each instance with its own local configuration variable.

The ordering used to look up a variable, called `<parameter name>`:

1. `<subsystem name>.<local name>.<parameter name>`

2. <local name>.<parameter name>
3. <subsystem name>.<parameter name>
4. <parameter name>

If this local name is not specified on the command line, numbers 1 and 2 are skipped. As soon as the first match is found, the search is completed, and the corresponding value is used.

This example configures a *condor_master* to run 2 *condor_schedd* daemons. The *condor_master* daemon needs the configuration:

```
XYZZY           = $(SCHEDD)
XYZZY_ARGS      = -local-name xyzzy
DAEMON_LIST     = $(DAEMON_LIST) XYZZY
DC_DAEMON_LIST  = + XYZZY
XYZZY_LOG       = $(LOG)/SchedLog.xyzzy
```

Using this example configuration, the *condor_master* starts up a second *condor_schedd* daemon, where this second *condor_schedd* daemon is passed **-local-name** *xyzzy* on the command line.

Continuing the example, configure the *condor_schedd* daemon named `xyzzy`. This *condor_schedd* daemon will share all configuration variable definitions with the other *condor_schedd* daemon, except for those specified separately.

```
SCHEDD.XYZZY.SCHEDD_NAME = XYZZY
SCHEDD.XYZZY.SCHEDD_LOG  = $(XYZZY_LOG)
SCHEDD.XYZZY.SPOOL       = $(SPOOL).XYZZY
```

Note that the example `SCHEDD_NAME` and `SPOOL` are specific to the *condor_schedd* daemon, as opposed to a different daemon such as the *condor_startd*. Other Condor daemons using this feature will have different requirements for which parameters need to be specified individually. This example works for the *condor_schedd*, and more local configuration can, and likely would be specified.

Also note that each daemon's log file must be specified individually, and in two places: one specification is for use by the *condor_master*, and the other is for use by the daemon itself. In the example, the `XYZZY` *condor_schedd* configuration variable `SCHEDD.XYZZY.SCHEDD_LOG` definition references the *condor_master* daemon's `XYZZY_LOG`.

Comments and Line Continuations

A Condor configuration file may contain comments and line continuations. A comment is any line beginning with a pound character (#). A continuation is any entry that continues across multiples lines. Line continuation is accomplished by placing the backslash character (\) at the end of any line to be continued onto another. Valid examples of line continuation are

```
START = (KeyboardIdle > 15 * $(MINUTE)) && \
((LoadAvg - CondorLoadAvg) <= 0.3)
```

and

```
ADMIN_MACHINES = condor.cs.wisc.edu, raven.cs.wisc.edu, \
stork.cs.wisc.edu, ostrich.cs.wisc.edu, \
bigbird.cs.wisc.edu
HOSTALLOW_ADMIN = $(ADMIN_MACHINES)
```

Note that a line continuation character may currently be used within a comment, so the following example does *not* set the configuration variable `FOO`:

```
# This comment includes the following line, so FOO is NOT set \
FOO = BAR
```

It is a poor idea to use this functionality, as it is likely to stop working in future Condor releases.

Executing a Program to Produce Configuration Macros

Instead of reading from a file, Condor may run a program to obtain configuration macros. The vertical bar character (`|`) as the last character defining a file name provides the syntax necessary to tell Condor to run a program. This syntax may only be used in the definition of the `CONDOR_CONFIG` environment variable, or the `LOCAL_CONFIG_FILE` configuration variable.

The command line for the program is formed by the characters preceding the vertical bar character. The standard output of the program is parsed as a configuration file would be.

An example:

```
LOCAL_CONFIG_FILE = /bin/make_the_config|
```

Program */bin/make_the_config* is executed, and its output is the set of configuration macros.

Note that either a program is executed to generate the configuration macros or the configuration is read from one or more files. The syntax uses space characters to separate command line elements, if an executed program produces the configuration macros. Space characters would otherwise separate the list of files. This syntax does not permit distinguishing one from the other, so only one may be specified.

Macros That Will Require a Restart When Changed

When any of the following listed configuration variables are changed, Condor must be restarted. Reconfiguration using *condor_reconfig* will not be enough.

- BIND_ALL_INTERFACES
- DAEMON_LIST
- FetchWorkDelay
- MAX_NUM_CPUS
- MAX_TRACKING_GID
- MIN_TRACKING_GID
- NETWORK_INTERFACE
- NUM_CPUS
- PREEMPTION_REQUIREMENTS_STABLE
- PRIVSEP_ENABLED
- PROCD_ADDRESS

Pre-Defined Macros

Condor provides pre-defined macros that help configure Condor. Pre-defined macros are listed as $(macro_name).

This first set are entries whose values are determined at run time and cannot be overwritten. These are inserted automatically by the library routine which parses the configuration files. This implies that a change to the underlying value of any of these variables will require a full restart of Condor in order to use the changed value.

$(FULL_HOSTNAME) The fully qualified host name of the local machine, which is host name plus domain name.

$(HOSTNAME) The host name of the local machine (no domain name).

$(IP_ADDRESS) The ASCII string version of the local machine's IP address.

$(TILDE) The full path to the home directory of the Unix user condor, if such a user exists on the local machine.

$(SUBSYSTEM) The subsystem name of the daemon or tool that is evaluating the macro. This is a unique string which identifies a given daemon within the Condor system. The possible subsystem names are:

- AMAZON_GAHP
- C_GAHP
- CKPT_SERVER

- COLLECTOR
- DBMSD
- GRIDMANAGER
- HAD
- HDFS
- JOB_ROUTER
- KBDD
- LEASEMANAGER
- MASTER
- NEGOTIATOR
- QUILL
- REPLICATION
- ROOSTER
- SCHEDD
- SHADOW
- STARTD
- STARTER
- SUBMIT
- TOOL
- TRANSFERER

This second set of macros are entries whose default values are determined automatically at run time but which can be overwritten.

$(ARCH) Defines the string used to identify the architecture of the local machine to Condor. The *condor_startd* will advertise itself with this attribute so that users can submit binaries compiled for a given platform and force them to run on the correct machines. *condor_submit* will append a requirement to the job ClassAd that it must run on the same ARCH and OPSYS of the machine where it was submitted, unless the user specifies ARCH and/or OPSYS explicitly in their submit file. See the the *condor_submit* manual page on page 825 for details.

$(OPSYS) Defines the string used to identify the operating system of the local machine to Condor. If it is not defined in the configuration file, Condor will automatically insert the operating system of this machine as determined by *uname*.

$(UNAME_ARCH) The architecture as reported by *uname*(2)'s machine field. Always the same as ARCH on Windows.

$(UNAME_OPSYS) The operating system as reported by *uname*(2)'s sysname field. Always the same as OPSYS on Windows.

$(DETECTED_MEMORY) The amount of detected physical memory (RAM) in Mbytes.

$(DETECTED_CORES) The number of detected CPU cores. This includes hyper threaded cores, if there are any.

$(PID) The process ID for the daemon or tool.

$(PPID) The process ID of the parent process for the daemon or tool.

$(USERNAME) The user name of the UID of the daemon or tool. For daemons started as root, but running under another UID (typically the user condor), this will be the other UID.

$(FILESYSTEM_DOMAIN) Defaults to the fully qualified host name of the machine it is evaluated on. See section 3.3.7, Shared File System Configuration File Entries for the full description of its use and under what conditions you would want to change it.

$(UID_DOMAIN) Defaults to the fully qualified host name of the machine it is evaluated on. See section 3.3.7 for the full description of this configuration variable.

Since `$(ARCH)` and `$(OPSYS)` will automatically be set to the correct values, we recommend that you do not overwrite them. Only do so if you know what you are doing.

3.3.2 Special Macros

References to the Condor process's environment are allowed in the configuration files. Environment references use the `ENV` macro and are of the form:

```
$ENV(environment_variable_name)
```

For example,

```
A = $ENV(HOME)
```

binds `A` to the value of the HOME environment variable. Environment references are not currently used in standard Condor configurations. However, they can sometimes be useful in custom configurations.

This same syntax is used in the `RANDOM_CHOICE()` macro to allow a random choice of a parameter within a configuration file. These references are of the form:

```
$RANDOM_CHOICE(list of parameters)
```

This allows a random choice within the parameter list to be made at configuration time. Of the list of parameters, one is chosen when encountered during configuration. For example, if one of the integers 0-8 (inclusive) should be randomly chosen, the macro usage is

```
$RANDOM_CHOICE(0,1,2,3,4,5,6,7,8)
```

The RANDOM_INTEGER() macro is similar to the RANDOM_CHOICE() macro, and is used to select a random integer within a configuration file. References are of the form:

```
$RANDOM_INTEGER(min, max [, step])
```

A random integer within the range min and max, inclusive, is selected at configuration time. The optional step parameter controls the stride within the range, and it defaults to the value 1. For example, to randomly chose an even integer in the range 0-8 (inclusive), the macro usage is

```
$RANDOM_INTEGER(0, 8, 2)
```

See section 7.2 on page 616 for an actual use of this specialized macro.

3.3.3 Condor-wide Configuration File Entries

This section describes settings which affect all parts of the Condor system. Other system-wide settings can be found in section 3.3.6 on "Network-Related Configuration File Entries", and section 3.3.7 on "Shared File System Configuration File Entries".

CONDOR_HOST This macro may be used to define the $(NEGOTIATOR_HOST) and is used to define the $(COLLECTOR_HOST) macro. Normally the *condor_collector* and *condor_negotiator* would run on the same machine. If for some reason they were not run on the same machine, $(CONDOR_HOST) would not be needed. Some of the host-based security macros use $(CONDOR_HOST) by default. See section 3.6.9, on Setting up IP/host-based security in Condor for details.

COLLECTOR_HOST The host name of the machine where the *condor_collector* is running for your pool. Normally, it is defined relative to the $(CONDOR_HOST) macro. There is no default value for this macro; COLLECTOR_HOST must be defined for the pool to work properly.

In addition to defining the host name, this setting can optionally be used to specify the network port of the *condor_collector*. The port is separated from the host name by a colon (':'). For example,

```
COLLECTOR_HOST = $(CONDOR_HOST):1234
```

If no port is specified, the default port of 9618 is used. Using the default port is recommended for most sites. It is only changed if there is a conflict with another service listening on the same network port. For more information about specifying a non-standard port for the *condor_collector* daemon, see section 3.7.1 on page 360.

NEGOTIATOR_HOST This configuration variable is no longer used. It previously defined the host name of the machine where the *condor_negotiator* is running. At present, the port where the *condor_negotiator* is listening is dynamically allocated.

CONDOR_VIEW_HOST The host name of the machine, optionally appended by a colon and the port number, where the CondorView server is running. This service is optional, and requires additional configuration to enable it. There is no default value for `CONDOR_VIEW_HOST`. If `CONDOR_VIEW_HOST` is not defined, no CondorView server is used. See section 3.13.7 on page 440 for more details.

SCHEDD_HOST The host name of the machine where the *condor_schedd* is running for your pool. This is the host that queues submitted jobs. Note that, in most condor installations, there is a *condor_schedd* running on each host from which jobs are submitted. The default value of `SCHEDD_HOST` is the current host. For most pools, this macro is not defined.

RELEASE_DIR The full path to the Condor release directory, which holds the `bin`, `etc`, `lib`, and `sbin` directories. Other macros are defined relative to this one. There is no default value for `RELEASE_DIR`.

BIN This directory points to the Condor directory where user-level programs are installed. It is usually defined relative to the `$(RELEASE_DIR)` macro. There is no default value for `BIN`.

LIB This directory points to the Condor directory where libraries used to link jobs for Condor's standard universe are stored. The *condor_compile* program uses this macro to find these libraries, so it must be defined for *condor_compile* to function. `$(LIB)` is usually defined relative to the `$(RELEASE_DIR)` macro, and has no default value.

LIBEXEC This directory points to the Condor directory where support commands that Condor needs will be placed. Do not add this directory to a user or system-wide path.

INCLUDE This directory points to the Condor directory where header files reside. `$(INCLUDE)` would usually be defined relative to the `$(RELEASE_DIR)` configuration macro. There is no default value, but if defined, it can make inclusion of necessary header files for compilation of programs (such as those programs that use `libcondorapi.a`) easier through the use of *condor_config_val*.

SBIN This directory points to the Condor directory where Condor's system binaries (such as the binaries for the Condor daemons) and administrative tools are installed. Whatever directory `$(SBIN)` points to ought to be in the `PATH` of users acting as Condor administrators. `SBIN` has no default value.

LOCAL_DIR The location of the local Condor directory on each machine in your pool. One common option is to use the condor user's home directory which may be specified with `$(TILDE)`. There is no default value for `LOCAL_DIR`. For example:

```
LOCAL_DIR = $(tilde)
```

On machines with a shared file system, where either the $(TILDE) directory or another directory you want to use is shared among all machines in your pool, you might use the $(HOSTNAME) macro and have a directory with many subdirectories, one for each machine in your pool, each named by host names. For example:

```
LOCAL_DIR = $(tilde)/hosts/$(hostname)
```

or:

```
LOCAL_DIR = $(release_dir)/hosts/$(hostname)
```

LOG Used to specify the directory where each Condor daemon writes its log files. The names of the log files themselves are defined with other macros, which use the $(LOG) macro by default. The log directory also acts as the current working directory of the Condor daemons as the run, so if one of them should produce a core file for any reason, it would be placed in the directory defined by this macro. LOG is required to be defined. Normally, $(LOG) is defined in terms of $(LOCAL_DIR).

SPOOL The spool directory is where certain files used by the *condor_schedd* are stored, such as the job queue file and the initial executables of any jobs that have been submitted. In addition, for systems not using a checkpoint server, all the checkpoint files from jobs that have been submitted from a given machine will be store in that machine's spool directory. Therefore, you will want to ensure that the spool directory is located on a partition with enough disk space. If a given machine is only set up to execute Condor jobs and not submit them, it would not need a spool directory (or this macro defined). There is no default value for SPOOL , and the *condor_schedd* will not function without it SPOOL defined. Normally, $(SPOOL) is defined in terms of $(LOCAL_DIR).

EXECUTE This directory acts as a place to create the scratch directory of any Condor job that is executing on the local machine. The scratch directory is the destination of any input files that were specified for transfer. It also serves as the job's working directory if the job is using file transfer mode and no other working directory was specified. If a given machine is set up to only submit jobs and not execute them, it would not need an execute directory, and this macro need not be defined. There is no default value for EXECUTE, and the *condor_startd* will not function if EXECUTE is undefined. Normally, $(EXECUTE) is defined in terms of $(LOCAL_DIR). To customize the execute directory independently for each batch slot, use SLOT<N>_EXECUTE.

SLOT<N>_EXECUTE Specifies an execute directory for use by a specific batch slot. <N> represents the number of the batch slot, such as 1, 2, 3, etc. This execute directory serves the same purpose as EXECUTE , but it allows the configuration of the directory independently for each batch slot. Having slots each using a different partition would be useful, for example, in preventing one job from filling up the same disk that other jobs are trying to write to. If this parameter is undefined for a given batch slot, it will use EXECUTE as the default. Note that each slot will advertise TotalDisk and Disk for the partition containing its execute directory.

LOCAL_CONFIG_FILE Identifies the location of the local, machine-specific configuration file for each machine in the pool. The two most common choices would be putting this file in the `$(LOCAL_DIR)`, or putting all local configuration files for the pool in a shared directory, each one named by host name. For example,

```
LOCAL_CONFIG_FILE = $(LOCAL_DIR)/condor_config.local
```

or,

```
LOCAL_CONFIG_FILE = $(release_dir)/etc/$(hostname).local
```

or, not using the release directory

```
LOCAL_CONFIG_FILE = /full/path/to/configs/$(hostname).local
```

The value of `LOCAL_CONFIG_FILE` is treated as a list of files, not a single file. The items in the list are delimited by either commas or space characters. This allows the specification of multiple files as the local configuration file, each one processed in the order given (with parameters set in later files overriding values from previous files). This allows the use of one global configuration file for multiple platforms in the pool, defines a platform-specific configuration file for each platform, and uses a local configuration file for each machine. If the list of files is changed in one of the later read files, the new list replaces the old list, but any files that have already been processed remain processed, and are removed from the new list if they are present to prevent cycles. See section 3.3.1 on page 158 for directions on using a program to generate the configuration macros that would otherwise reside in one or more files as described here. If `LOCAL_CONFIG_FILE` is not defined, no local configuration files are processed. For more information on this, see section 3.13.4 about Configuring Condor for Multiple Platforms on page 435.

If all files in a directory are local configuration files to be processed, then consider using `LOCAL_CONFIG_DIR`, defined at section 3.3.3.

REQUIRE_LOCAL_CONFIG_FILE A boolean value that defaults to `True`. When `True`, Condor exits with an error, if any file listed in `LOCAL_CONFIG_FILE` cannot be read. A value of `False` allows local configuration files to be missing. This is most useful for sites that have both large numbers of machines in the pool and a local configuration file that uses the `$(HOSTNAME)` macro in its definition. Instead of having an empty file for every host in the pool, files can simply be omitted.

LOCAL_CONFIG_DIR A directory may be used as a container for local configuration files. The files found in the directory are sorted into lexicographical order by file name, and then each file is treated as though it was listed in `LOCAL_CONFIG_FILE`. `LOCAL_CONFIG_DIR` is processed before any files listed in `LOCAL_CONFIG_FILE`, and is checked again after processing the `LOCAL_CONFIG_FILE` list. It is a list of directories, and each directory is processed in the order it appears in the list. The process is not recursive,

so any directories found inside the directory being processed are ignored. See also LOCAL_CONFIG_DIR_EXCLUDE_REGEXP.

LOCAL_CONFIG_DIR_EXCLUDE_REGEXP A regular expression that specifies file names to be ignored when looking for configuration files within the directories specified via LOCAL_CONFIG_DIR. The default expression ignores files with names beginning with a '.' or a '#', as well as files with names ending in '~'. This avoids accidents that can be caused by treating temporary files created by text editors as configuration files.

CONDOR_IDS The User ID (UID) and Group ID (GID) pair that the Condor daemons should run as, if the daemons are spawned as root. This value can also be specified in the CONDOR_IDS environment variable. If the Condor daemons are not started as root, then neither this CONDOR_IDS configuration macro nor the CONDOR_IDS environment variable are used. The value is given by two integers, separated by a period. For example, CONDOR_IDS = 1234.1234. If this pair is not specified in either the configuration file or in the environment, and the Condor daemons are spawned as root, then Condor will search for a condor user on the system, and run as that user's UID and GID. See section 3.6.13 on UIDs in Condor for more details.

CONDOR_ADMIN The email address that Condor will send mail to if something goes wrong in the pool. For example, if a daemon crashes, the *condor_master* can send an *obituary* to this address with the last few lines of that daemon's log file and a brief message that describes what signal or exit status that daemon exited with. There is no default value for CONDOR_ADMIN.

<SUBSYS>_ADMIN_EMAIL The email address that Condor will send mail to if something goes wrong with the named <SUBSYS>. Identical to CONDOR_ADMIN, but done on a per subsystem basis. There is no default value.

CONDOR_SUPPORT_EMAIL The email address to be included at the bottom of all email Condor sends out under the label "Email address of the local Condor administrator:". This is the address where Condor users at your site should send their questions about Condor and get technical support. If this setting is not defined, Condor will use the address specified in CONDOR_ADMIN (described above).

EMAIL_SIGNATURE Every e-mail sent by Condor includes a short signature line appended to the body. By default, this signature includes the URL to the global Condor project website. When set, this variable defines an alternative signature line to be used instead of the default. Note that the value can only be one line in length. This variable could be used to direct users to look at local web site with information specific to the installation of Condor.

MAIL The full path to a mail sending program that uses **-s** to specify a subject for the message. On all platforms, the default shipped with Condor should work. Only if you installed things in a non-standard location on your system would you need to change this setting. There is no default value for MAIL, and the *condor_schedd* will not function unless MAIL is defined.

RESERVED_SWAP The amount of swap space in Mbytes to reserve for this machine. Condor will not start up more *condor_shadow* processes if the amount of free swap space on this machine falls below this level. The default value is 0, which disables this check. It is anticipated that this configuration variable will no longer be used in the near future. If RESERVED_SWAP is *not* set to 0, the value of SHADOW_SIZE_ESTIMATE is used.

RESERVED_DISK Determines how much disk space you want to reserve for your own machine. When Condor is reporting the amount of free disk space in a given partition on your machine, it will always subtract this amount. An example is the *condor_startd*, which advertises the amount of free space in the `$(EXECUTE)` directory. The default value of `RESERVED_DISK` is zero.

LOCK Condor needs to create lock files to synchronize access to various log files. Because of problems with network file systems and file locking over the years, we *highly* recommend that you put these lock files on a local partition on each machine. If you do not have your `$(LOCAL_DIR)` on a local partition, be sure to change this entry.

Whatever user or group Condor is running as needs to have write access to this directory. If you are not running as root, this is whatever user you started up the *condor_master* as. If you are running as root, and there is a condor account, it is most likely condor. Otherwise, it is whatever you set in the `CONDOR_IDS` environment variable, or whatever you define in the `CONDOR_IDS` setting in the Condor config files. See section 3.6.13 on UIDs in Condor for details.

If no value for `LOCK` is provided, the value of `LOG` is used.

HISTORY Defines the location of the Condor history file, which stores information about all Condor jobs that have completed on a given machine. This macro is used by both the *condor_schedd* which appends the information and *condor_history*, the user-level program used to view the history file. This configuration macro is given the default value of `$(SPOOL)/history` in the default configuration. If not defined, no history file is kept.

ENABLE_HISTORY_ROTATION If this is defined to be true, then the history file will be rotated. If it is false, then it will not be rotated, and it will grow indefinitely, to the limits allowed by the operating system. If this is not defined, it is assumed to be true. The rotated files will be stored in the same directory as the history file.

MAX_HISTORY_LOG Defines the maximum size for the history file, in bytes. It defaults to 20MB. This parameter is only used if history file rotation is enabled.

MAX_HISTORY_ROTATIONS When history file rotation is turned on, this controls how many backup files there are. It default to 2, which means that there may be up to three history files (two backups, plus the history file that is being currently written to). When the history file is rotated, and this rotation would cause the number of backups to be too large, the oldest file is removed.

MAX_JOB_QUEUE_LOG_ROTATIONS The schedd periodically rotates the job queue database file in order to save disk space. This option controls how many rotated files are saved. It defaults to 1, which means there may be up to two history files (the previous one, which was rotated out of use, and the current one that is being written to). When the job queue file is rotated, and this rotation would cause the number of backups to be larger the the maximum specified, the oldest file is removed. The primary reason to save one or more rotated job queue files is if you are using Quill, and you want to ensure that Quill keeps an accurate history of all events logged in the job queue file. Quill keeps track of where it last left off when reading logged events, so when the file is rotated, Quill will resume reading from where it last left

off, provided that the rotated file still exists. If Quill finds that it needs to read events from a rotated file that has been deleted, it will be forced to skip the missing events and resume reading in the next chronological job queue file that can be found. Such an event should not lead to an inconsistency in Quill's view of the current queue contents, but it would create a inconsistency in Quill's record of the history of the job queue.

DEFAULT_DOMAIN_NAME The value to be appended to a machine's host name, representing a domain name, which Condor then uses to form a fully qualified host name. This is required if there is no fully qualified host name in file `/etc/hosts` or in NIS. Set the value in the global configuration file, as Condor may depend on knowing this value in order to locate the local configuration file(s). The default value as given in the sample configuration file of the Condor download is bogus, and must be changed. If this variable is removed from the global configuration file, or if the definition is empty, then Condor attempts to discover the value.

NO_DNS A boolean value that defaults to `False`. When `True`, Condor constructs host names using the host's IP address together with the value defined for `DEFAULT_DOMAIN_NAME`.

CM_IP_ADDR If neither `COLLECTOR_HOST` nor `COLLECTOR_IP_ADDR` macros are defined, then this macro will be used to determine the IP address of the central manager (collector daemon). This macro is defined by an IP address.

EMAIL_DOMAIN By default, if a user does not specify `notify_user` in the submit description file, any email Condor sends about that job will go to "username@UID_DOMAIN". If your machines all share a common UID domain (so that you would set `UID_DOMAIN` to be the same across all machines in your pool), but email to user@UID_DOMAIN is not the right place for Condor to send email for your site, you can define the default domain to use for email. A common example would be to set `EMAIL_DOMAIN` to the fully qualified host name of each machine in your pool, so users submitting jobs from a specific machine would get email sent to user@machine.your.domain, instead of user@your.domain. You would do this by setting `EMAIL_DOMAIN` to `$(FULL_HOSTNAME)`. In general, you should leave this setting commented out unless two things are true: 1) `UID_DOMAIN` is set to your domain, not `$(FULL_HOSTNAME)`, and 2) email to user@UID_DOMAIN will not work.

CREATE_CORE_FILES Defines whether or not Condor daemons are to create a core file in the `LOG` directory if something really bad happens. It is used to set the resource limit for the size of a core file. If not defined, it leaves in place whatever limit was in effect when the Condor daemons (normally the *condor_master*) were started. This allows Condor to inherit the default system core file generation behavior at start up. For Unix operating systems, this behavior can be inherited from the parent shell, or specified in a shell script that starts Condor. If this parameter is set and `True`, the limit is increased to the maximum. If it is set to `False`, the limit is set at 0 (which means that no core files are created). Core files greatly help the Condor developers debug any problems you might be having. By using the parameter, you do not have to worry about tracking down where in your boot scripts you need to set the core limit before starting Condor. You set the parameter to whatever behavior you want Condor to enforce. This parameter defaults to undefined to allow the initial operating system default value to take precedence, and is commented out in the default configuration file.

CKPT_PROBE Defines the path and executable name of the helper process Condor will use to determine information for the `CheckpointPlatform` attribute in the machine's ClassAd. The default value is `$(LIBEXEC)/condor_ckpt_probe`.

ABORT_ON_EXCEPTION When Condor programs detect a fatal internal exception, they normally log an error message and exit. If you have turned on `CREATE_CORE_FILES`, in some cases you may also want to turn on `ABORT_ON_EXCEPTION` so that core files are generated when an exception occurs. Set the following to True if that is what you want.

Q_QUERY_TIMEOUT Defines the timeout (in seconds) that *condor_q* uses when trying to connect to the *condor_schedd*. Defaults to 20 seconds.

DEAD_COLLECTOR_MAX_AVOIDANCE_TIME Defines the interval of time (in seconds) between checks for a failed primary *condor_collector* daemon. If connections to the dead primary *condor_collector* take very little time to fail, new attempts to query the primary *condor_collector* may be more frequent than the specified maximum avoidance time. The default value equals one hour. This variable has relevance to flocked jobs, as it defines the maximum time they may be reporting to the primary *condor_collector* without the *condor_negotiator* noticing.

PASSWD_CACHE_REFRESH Condor can cause NIS servers to become overwhelmed by queries for uid and group information in large pools. In order to avoid this problem, Condor caches UID and group information internally. This integer value allows pool administrators to specify (in seconds) how long Condor should wait until refreshes a cache entry. The default is set to 300 seconds, or 5 minutes, plus a random number of seconds between 0 and 60 to avoid having lots of processes refreshing at the same time. This means that if a pool administrator updates the user or group database (for example, `/etc/passwd` or `/etc/group`), it can take up to 6 minutes before Condor will have the updated information. This caching feature can be disabled by setting the refresh interval to 0. In addition, the cache can also be flushed explicitly by running the command *condor_reconfig*. This configuration variable has no effect on Windows.

SYSAPI_GET_LOADAVG If set to False, then Condor will not attempt to compute the load average on the system, and instead will always report the system load average to be 0.0. Defaults to True.

NETWORK_MAX_PENDING_CONNECTS This specifies a limit to the maximum number of simultaneous network connection attempts. This is primarily relevant to *condor_schedd*, which may try to connect to large numbers of startds when claiming them. The negotiator may also connect to large numbers of startds when initiating security sessions used for sending MATCH messages. On Unix, the default for this parameter is eighty percent of the process file descriptor limit. On windows, the default is 1600.

WANT_UDP_COMMAND_SOCKET This setting, added in version 6.9.5, controls if Condor daemons should create a UDP command socket in addition to the TCP command socket (which is required). The default is `True`, and modifying it requires restarting all Condor daemons, not just a *condor_reconfig* or SIGHUP.

Normally, updates sent to the *condor_collector* use UDP, in addition to certain keep alive messages and other non-essential communication. However, in certain situations, it might

be desirable to disable the UDP command port (for example, to reduce the number of ports represented by a GCB broker, etc).

Unfortunately, due to a limitation in how these command sockets are created, it is not possible to define this setting on a per-daemon basis, for example, by trying to set `STARTD.WANT_UDP_COMMAND_SOCKET`. At least for now, this setting must be defined machine wide to function correctly.

If this setting is set to true on a machine running a *condor_collector*, the pool should be configured to use TCP updates to that collector (see section 3.7.6 on page 383 for more information).

ALLOW_SCRIPTS_TO_RUN_AS_EXECUTABLES A boolean value that, when `True`, permits scripts on Windows platforms to be used in place of the **executable** in a job submit description file, in place of a *condor_dagman* pre or post script, or in producing the configuration, for example. Allows a script to be used in any circumstance previously limited to a Windows executable or a batch file. The default value is `True`. See section 6.2.7 on page 599 for further description.

OPEN_VERB_FOR_<EXT>_FILES A string that defines a Windows *verb* for use in a root hive registry look up. `<EXT>` defines the file name extension, which represents a scripting language, also needed for the look up. See section 6.2.7 on page 599 for a more complete description.

STRICT_CLASSAD_EVALUATION A boolean value that controls how ClassAd expressions are evaluated. If set to `True`, then New ClassAd evaluation semantics are used. This means that attribute references without a `MY.` or `TARGET.` prefix are only looked up in the local ClassAd. If set to the default value of `False`, Old ClassAd evaluation semantics are used. See section 4.1.1 on page 476 for details.

CLASSAD_USER_LIBS A comma separated list of paths to shared libraries that contain additional ClassAd functions to be used during ClassAd evaluation.

3.3.4 Daemon Logging Configuration File Entries

These entries control how and where the Condor daemons write to log files. Many of the entries in this section represents multiple macros. There is one for each subsystem (listed in section 3.3.1). The macro name for each substitutes `<SUBSYS>` with the name of the subsystem corresponding to the daemon.

<SUBSYS>_LOG The name of the log file for a given subsystem. For example, `$(STARTD_LOG)` gives the location of the log file for *condor_startd*. The default is `$(LOG)/<SUBSYS>LOG`.

MAX_<SUBSYS>_LOG Controls the maximum length in bytes to which a log will be allowed to grow. Each log file will grow to the specified length, then be saved to a file with an ISO timestamp suffix. The oldest rotated file receives the ending `.old`. The `.old` files are overwritten each time the maximum number of rotated files (determined by the value of `MAX_NUM_<SUBSYS>_LOG`) is exceeded. Thus, the maximum space devoted to logging for

any one program will be MAX_NUM_<SUBSYS>_LOG + 1 times the maximum length of its log file. A value of 0 specifies that the file may grow without bounds. The default is 1 Mbyte.

MAX_NUM_<SUBSYS>_LOG An integer that controls the maximum number of rotations a log file is allowed to perform before the oldest one will be rotated away. Thus, at most MAX_NUM_<SUBSYS>_LOG + 1 log files of the same program coexist at a given time. The default value is 1.

TRUNC_<SUBSYS>_LOG_ON_OPEN If this macro is defined and set to True, the affected log will be truncated and started from an empty file with each invocation of the program. Otherwise, new invocations of the program will append to the previous log file. By default this setting is False for all daemons.

<SUBSYS>_LOCK This macro specifies the lock file used to synchronize append operations to the log file for this subsystem. It must be a separate file from the $(<SUBSYS>_LOG) file, since the $(<SUBSYS>_LOG) file may be rotated and you want to be able to synchronize access across log file rotations. A lock file is only required for log files which are accessed by more than one process. Currently, this includes only the SHADOW subsystem. This macro is defined relative to the $(LOCK) macro.

FILE_LOCK_VIA_MUTEX This macro setting only works on Win32 – it is ignored on Unix. If set to be True, then log locking is implemented via a kernel mutex instead of via file locking. On Win32, mutex access is FIFO, while obtaining a file lock is non-deterministic. Thus setting to True fixes problems on Win32 where processes (usually shadows) could starve waiting for a lock on a log file. Defaults to True on Win32, and is always False on Unix.

LOCK_DEBUG_LOG_TO_APPEND A boolean value that defaults to False. This variable controls whether a daemon's debug lock is used when appending to the log. When False, the debug lock is only used when rotating the log file. This is more efficient, especially when many processes share the same log file. When True, the debug lock is used when writing to the log, as well as when rotating the log file. This setting is ignored under Windows, and the behavior of Windows platforms is as though this variable were True. Under Unix, the default value of False is appropriate when logging to file systems that support the POSIX semantics of O_APPEND. On non-POSIX-compliant file systems, it is possible for the characters in log messages from multiple processes sharing the same log to be interleaved, unless locking is used. Since Condor does not support sharing of debug logs between processes running on different machines, many non-POSIX-compliant file systems will still avoid interleaved messages without requiring Condor to use a lock. Tests of AFS and NFS have not revealed any problems when appending to the log without locking.

ENABLE_USERLOG_LOCKING When True (the default value), a user's job log (as specified in a submit description file) will be locked before being written to. If False, Condor will not lock the file before writing.

CREATE_LOCKS_ON_LOCAL_DISK A boolean value utilized only for Unix operating systems, that defaults to True. This variable is only relevant if ENABLE_USERLOG_LOCKING is True. When True, job user logs and the global job event log are written to a directory named condorLocks, thereby using a local drive to avoid known problems with locking on NFS. The location of the condorLocks directory is determined by

1. The value of TEMP_DIR, if defined.
2. The value of TMP_DIR, if defined and TEMP_DIR is not defined.
3. The default value of /tmp, if neither TEMP_DIR nor TMP_DIR is defined.

TOUCH_LOG_INTERVAL The time interval in seconds between when daemons touch their log files. The change in last modification time for the log file is useful when a daemon restarts after failure or shut down. The last modification date is printed, and it provides an upper bound on the length of time that the daemon was not running. Defaults to 60 seconds.

LOGS_USE_TIMESTAMP This macro controls how the current time is formatted at the start of each line in the daemon log files. When True, the Unix time is printed (number of seconds since 00:00:00 UTC, January 1, 1970). When False (the default value), the time is printed like so: <Month>/<Day> <Hour>:<Minute>:<Second> in the local timezone.

DEBUG_TIME_FORMAT This string defines how to format the current time printed at the start of each line in the daemon log files. The value is a format string is passed to the C strftime() function, so see that manual page for platform-specific details. If not defined, the default value is

```
"%m/%d %H:%M:%S "
```

<SUBSYS>_DEBUG All of the Condor daemons can produce different levels of output depending on how much information is desired. The various levels of verbosity for a given daemon are determined by this macro. All daemons have the default level D_ALWAYS, and log messages for that level will be printed to the daemon's log, regardless of this macro's setting. Settings are a comma- or space-separated list of the following values:

D_ALL This flag turns on *all* debugging output by enabling all of the debug levels at once. There is no need to list any other debug levels in addition to D_ALL; doing so would be redundant. Be warned: this will generate about a *HUGE* amount of output. To obtain a higher level of output than the default, consider using D_FULLDEBUG before using this option.

D_FULLDEBUG This level provides verbose output of a general nature into the log files. Frequent log messages for very specific debugging purposes would be excluded. In those cases, the messages would be viewed by having that another flag and D_FULLDEBUG both listed in the configuration file.

D_DAEMONCORE Provides log file entries specific to DaemonCore, such as timers the daemons have set and the commands that are registered. If both D_FULLDEBUG and D_DAEMONCORE are set, expect *very* verbose output.

D_PRIV This flag provides log messages about the *privilege state* switching that the daemons do. See section 3.6.13 on UIDs in Condor for details.

D_COMMAND With this flag set, any daemon that uses DaemonCore will print out a log message whenever a command comes in. The name and integer of the command, whether the command was sent via UDP or TCP, and where the command was sent from are all

logged. Because the messages about the command used by *condor_kbdd* to communicate with the *condor_startd* whenever there is activity on the X server, and the command used for keep-alives are both only printed with D_FULLDEBUG enabled, it is best if this setting is used for all daemons.

D_LOAD The *condor_startd* keeps track of the load average on the machine where it is running. Both the general system load average, and the load average being generated by Condor's activity there are determined. With this flag set, the *condor_startd* will log a message with the current state of both of these load averages whenever it computes them. This flag only affects the *condor_startd*.

D_KEYBOARD With this flag set, the *condor_startd* will print out a log message with the current values for remote and local keyboard idle time. This flag affects only the *condor_startd*.

D_JOB When this flag is set, the *condor_startd* will send to its log file the contents of any job ClassAd that the *condor_schedd* sends to claim the *condor_startd* for its use. This flag affects only the *condor_startd*.

D_MACHINE When this flag is set, the *condor_startd* will send to its log file the contents of its resource ClassAd when the *condor_schedd* tries to claim the *condor_startd* for its use. This flag affects only the *condor_startd*.

D_SYSCALLS This flag is used to make the *condor_shadow* log remote syscall requests and return values. This can help track down problems a user is having with a particular job by providing the system calls the job is performing. If any are failing, the reason for the failure is given. The *condor_schedd* also uses this flag for the server portion of the queue management code. With D_SYSCALLS defined in SCHEDD_DEBUG there will be verbose logging of all queue management operations the *condor_schedd* performs.

D_MATCH When this flag is set, the *condor_negotiator* logs a message for every match.

D_NETWORK When this flag is set, all Condor daemons will log a message on every TCP accept, connect, and close, and on every UDP send and receive. This flag is not yet fully supported in the *condor_shadow*.

D_HOSTNAME When this flag is set, the Condor daemons and/or tools will print verbose messages explaining how they resolve host names, domain names, and IP addresses. This is useful for sites that are having trouble getting Condor to work because of problems with DNS, NIS or other host name resolving systems in use.

D_CKPT When this flag is set, the Condor process checkpoint support code, which is linked into a STANDARD universe user job, will output some low-level details about the checkpoint procedure into the $(SHADOW_LOG).

D_SECURITY This flag will enable debug messages pertaining to the setup of secure network communication, including messages for the negotiation of a socket authentication mechanism, the management of a session key cache. and messages about the authentication process itself. See section 3.6.1 for more information about secure communication configuration.

D_PROCFAMILY Condor often times needs to manage an entire family of processes, (that is, a process and all descendants of that process). This debug flag will turn on debugging output for the management of families of processes.

D_ACCOUNTANT When this flag is set, the *condor_negotiator* will output debug messages relating to the computation of user priorities (see section 3.4).

D_PROTOCOL Enable debug messages relating to the protocol for Condor's matchmaking and resource claiming framework.

D_PID This flag is different from the other flags, because it is used to change the formatting of all log messages that are printed, as opposed to specifying what kinds of messages should be printed. If D_PID is set, Condor will always print out the process identifier (PID) of the process writing each line to the log file. This is especially helpful for Condor daemons that can fork multiple helper-processes (such as the *condor_schedd* or *condor_collector*) so the log file will clearly show which thread of execution is generating each log message.

D_FDS This flag is different from the other flags, because it is used to change the formatting of all log messages that are printed, as opposed to specifying what kinds of messages should be printed. If D_FDS is set, Condor will always print out the file descriptor that the open of the log file was allocated by the operating system. This can be helpful in debugging Condor's use of system file descriptors as it will generally track the number of file descriptors that Condor has open.

ALL_DEBUG Used to make all subsystems share a debug flag. Set the parameter ALL_DEBUG instead of changing all of the individual parameters. For example, to turn on all debugging in all subsystems, set ALL_DEBUG = D_ALL.

TOOL_DEBUG Uses the same values (debugging levels) as <SUBSYS>_DEBUG to describe the amount of debugging information sent to stderr for Condor tools.

Log files may optionally be specified per debug level as follows:

<SUBSYS>_<LEVEL>_LOG The name of a log file for messages at a specific debug level for a specific subsystem. <LEVEL> is defined by any debug level, but without the D_ prefix. See section 3.3.4 for the list of debug levels. If the debug level is included in $(<SUBSYS>_DEBUG), then all messages of this debug level will be written both to the log file defined by <SUBSYS>_LOG and the the log file defined by <SUBSYS>_<LEVEL>_LOG. As examples, SHADOW_SYSCALLS_LOG specifies a log file for all remote system call debug messages, and NEGOTIATOR_MATCH_LOG specifies a log file that only captures *condor_negotiator* debug events occurring with matches.

MAX_<SUBSYS>_<LEVEL>_LOG See section 3.3.4, the definition of MAX_<SUBSYS>_LOG.

TRUNC_<SUBSYS>_<LEVEL>_LOG_ON_OPEN Similar to TRUNC_<SUBSYS>_LOG_ON_OPEN .

The following macros control where and what is written to the event log, a file that receives job user log events, but across all users and user's jobs.

EVENT_LOG The full path and file name of the event log. There is no default value for this variable, so no event log will be written, if not defined.

EVENT_LOG_MAX_SIZE Controls the maximum length in bytes to which the event log will be allowed to grow. The log file will grow to the specified length, then be saved to a file with the suffix .old. The .old files are overwritten each time the log is saved. A value of 0 specifies that the file may grow without bounds (and disables rotation). The default is 1 Mbyte. For backwards compatibility, MAX_EVENT_LOG will be used if EVENT_LOG_MAX_SIZE is not defined. If EVENT_LOG is not defined, this parameter has no effect.

MAX_EVENT_LOG See EVENT_LOG_MAX_SIZE.

EVENT_LOG_MAX_ROTATIONS Controls the maximum number of rotations of the event log that will be stored. If this value is 1 (the default), the event log will be rotated to a ".old" file as described above. However, if this is greater than 1, then multiple rotation files will be stores, up to EVENT_LOG_MAX_ROTATIONS of them. These files will be named, instead of the ".old" suffix, ".1", ".2", with the ".1" being the most recent rotation. This is an integer parameter with a default value of 1. If EVENT_LOG is not defined, or if EVENT_LOG_MAX_SIZE has a value of 0 (which disables event log rotation), this parameter has no effect.

EVENT_LOG_ROTATION_LOCK Controls the lock file that will be used to ensure that, when rotating files, the rotation is done by a single process. This is a string parameter; it's default value is the file path of the event log itself, with a ".lock" appended. If EVENT_LOG is not defined, or if EVENT_LOG_MAX_SIZE has a value of 0 (which disables event log rotation), this parameter has no effect.

EVENT_LOG_FSYNC A boolean value that controls whether Condor will perform an `fsync()` after writing each event to the event log. When `True`, an `fsync()` operation is performed after each event. This `fsync()` operation forces the operating system to synchronize the updates to the event log to the disk, but can negatively affect the performance of the system. Defaults to `False`.

EVENT_LOG_LOCKING A boolean value that defaults to `True`. When `True`, the event log (as specified by EVENT_LOG) will be locked before being written to. When `False`, Condor does not lock the file before writing.

EVENT_LOG_USE_XML A boolean value that defaults to `False`. When `True`, events are logged in XML format. If EVENT_LOG is not defined, this parameter has no effect.

EVENT_LOG_JOB_AD_INFORMATION_ATTRS A comma-separated list of job ClassAd attributes, whose evaluated values form a new event, the JobAdInformationEvent. This new event is placed in the event log in addition to each logged event. If EVENT_LOG is not defined, this parameter has no effect. This configuration setting is the same as the job ad attribute `JobAdInformationAttrs` (see page 905) but it applies to the system event log rather than the user log.

3.3.5 DaemonCore Configuration File Entries

Please read section 3.9 for details on DaemonCore. There are certain configuration file settings that DaemonCore uses which affect all Condor daemons (except the checkpoint server, standard universe shadow, and standard universe starter, none of which use DaemonCore).

HOSTALLOW... All macros that begin with either HOSTALLOW or HOSTDENY are settings for Condor's host-based security. See section 3.6.9 on Setting up IP/host-based security in Condor for details on these macros and how to configure them.

ENABLE_RUNTIME_CONFIG The *condor_config_val* tool has an option **-rset** for dynamically setting run time configuration values, and which only affect the in-memory configuration variables. Because of the potential security implications of this feature, by default, Condor daemons will not honor these requests. To use this functionality, Condor administrators must specifically enable it by setting ENABLE_RUNTIME_CONFIG to True, and specify what configuration variables can be changed using the SETTABLE_ATTRS... family of configuration options. Defaults to False.

ENABLE_PERSISTENT_CONFIG The *condor_config_val* tool has a **-set** option for dynamically setting persistent configuration values. These values override options in the normal Condor configuration files. Because of the potential security implications of this feature, by default, Condor daemons will not honor these requests. To use this functionality, Condor administrators must specifically enable it by setting ENABLE_PERSISTENT_CONFIG to True, creating a directory where the Condor daemons will hold these dynamically-generated persistent configuration files (declared using PERSISTENT_CONFIG_DIR, described below) and specify what configuration variables can be changed using the SETTABLE_ATTRS... family of configuration options. Defaults to False.

PERSISTENT_CONFIG_DIR Directory where daemons should store dynamically-generated persistent configuration files (used to support *condor_config_val* **-set**) This directory should **only** be writable by root, or the user the Condor daemons are running as (if non-root). There is no default, administrators that wish to use this functionality must create this directory and define this setting. This directory must not be shared by multiple Condor installations, though it can be shared by all Condor daemons on the same host. Keep in mind that this directory should not be placed on an NFS mount where "root-squashing" is in effect, or else Condor daemons running as root will not be able to write to them. A directory (only writable by root) on the local file system is usually the best location for this directory.

SETTABLE_ATTRS... All macros that begin with SETTABLE_ATTRS or <SUBSYS>.SETTABLE_ATTRS are settings used to restrict the configuration values that can be changed using the *condor_config_val* command. Section 3.6.9 on Setting up IP/Host-Based Security in Condor for details on these macros and how to configure them. In particular, section 3.6.9 on page 342 contains details specific to these macros.

SHUTDOWN_GRACEFUL_TIMEOUT Determines how long Condor will allow daemons try their graceful shutdown methods before they do a hard shutdown. It is defined in terms of seconds. The default is 1800 (30 minutes).

<SUBSYS>_ADDRESS_FILE A complete path to a file that is to contain an IP address and port number for a daemon. Every Condor daemon that uses DaemonCore has a command port where commands are sent. The IP/port of the daemon is put in that daemon's ClassAd, so that other machines in the pool can query the *condor_collector* (which listens on a well-known port) to find the address of a given daemon on a given machine. When tools and daemons are all executing on the same single machine, communications do not require a query of the

condor_collector daemon. Instead, they look in a file on the local disk to find the IP/port. This macro causes daemons to write the IP/port of their command socket to a specified file. In this way, local tools will continue to operate, even if the machine running the *condor_collector* crashes. Using this file will also generate slightly less network traffic in the pool, since tools including *condor_q* and *condor_rm* do not need to send any messages over the network to locate the *condor_schedd* daemon. This macro is not necessary for the *condor_collector* daemon, since its command socket is at a well-known port.

The macro is named by substituting <SUBSYS> with the appropriate subsystem string as defined in section 3.3.1.

<SUBSYS>_DAEMON_AD_FILE A complete path to a file that is to contain the ClassAd for a daemon. When the daemon sends a ClassAd describing itself to the *condor_collector*, it will also place a copy of the ClassAd in this file. Currently, this setting only works for the *condor_schedd* (that is SCHEDD_DAEMON_AD_FILE) and is required for Quill.

<SUBSYS>_ATTRS or <SUBSYS>_EXPRS Allows any DaemonCore daemon to advertise arbitrary expressions from the configuration file in its ClassAd. Give the comma-separated list of entries from the configuration file you want in the given daemon's ClassAd. Frequently used to add attributes to machines so that the machines can discriminate between other machines in a job's **rank** and **requirements**.

The macro is named by substituting <SUBSYS> with the appropriate subsystem string as defined in section 3.3.1.

<SUBSYS>_EXPRS is a historic setting that functions identically to <SUBSYS>_ATTRS. Use <SUBSYS>_ATTRS.

NOTE: The *condor_kbdd* does not send ClassAds now, so this entry does not affect it. The *condor_startd*, *condor_schedd*, *condor_master*, and *condor_collector* do send ClassAds, so those would be valid subsystems to set this entry for.

SUBMIT_EXPRS not part of the <SUBSYS>_EXPRS, it is documented in section 3.3.14

Because of the different syntax of the configuration file and ClassAds, a little extra work is required to get a given entry into a ClassAd. In particular, ClassAds require quote marks (") around strings. Numeric values and boolean expressions can go in directly. For example, if the *condor_startd* is to advertise a string macro, a numeric macro, and a boolean expression, do something similar to:

```
STRING = This is a string
NUMBER = 666
BOOL1 = True
BOOL2 = CurrentTime >= $(NUMBER) || $(BOOL1)
MY_STRING = "$(STRING)"
STARTD_ATTRS = MY_STRING, NUMBER, BOOL1, BOOL2
```

DAEMON_SHUTDOWN Starting with Condor version 6.9.3, whenever a daemon is about to publish a ClassAd update to the *condor_collector*, it will evaluate this expression. If it evaluates to

`True`, the daemon will gracefully shut itself down, exit with the exit code 99, and will not be restarted by the *condor_master* (as if it sent itself a *condor_off* command). The expression is evaluated in the context of the ClassAd that is being sent to the *condor_collector*, so it can reference any attributes that can be seen with `condor_status -long [-daemon_type]` (for example, `condor_status -long [-master]` for the *condor_master*). Since each daemon's ClassAd will contain different attributes, administrators should define these shutdown expressions specific to each daemon, for example:

```
STARTD.DAEMON_SHUTDOWN = when to shutdown the startd
MASTER.DAEMON_SHUTDOWN = when to shutdown the master
```

Normally, these expressions would not be necessary, so if not defined, they default to FALSE. One possible use case is for Condor glide-in, to have the *condor_startd* shut itself down if it has not been claimed by a job after a certain period of time.

NOTE: This functionality does not work in conjunction with Condor's high-availability support (see section 3.11 on page 396 for more information). If you enable high-availability for a particular daemon, you should not define this expression.

DAEMON_SHUTDOWN_FAST Identical to `DAEMON_SHUTDOWN` (defined above), except the daemon will use the fast shutdown mode (as if it sent itself a *condor_off* command using the **-fast** option).

USE_CLONE_TO_CREATE_PROCESSES This setting controls how a Condor daemon creates a new process under certain versions of Linux. If set to `True` (the default value), the `clone` system call is used. Otherwise, the `fork` system call is used. `clone` provides scalability improvements for daemons using a large amount of memory (e.g. a *condor_schedd* with a lot of jobs in the queue). Currently, the use of `clone` is available on Linux systems other than IA-64, but not when GCB is enabled. If Condor detects that it is running under the valgrind analysis tools, this setting is ignored and treated as `False` to work around incompatibilities.

NOT_RESPONDING_TIMEOUT When a Condor daemon's parent process is another Condor daemon, the child daemon will periodically send a short message to its parent stating that it is alive and well. If the parent does not hear from the child for a while, the parent assumes that the child is hung, kills the child, and restarts the child. This parameter controls how long the parent waits before killing the child. It is defined in terms of seconds and defaults to 3600 (1 hour). The child sends its alive and well messages at an interval of one third of this value.

<SUBSYS>_NOT_RESPONDING_TIMEOUT Identical to `NOT_RESPONDING_TIMEOUT`, but controls the timeout for a specific type of daemon. For example, `SCHEDD_NOT_RESPONDING_TIMEOUT` controls how long the *condor_schedd*'s parent daemon will wait without receiving an alive and well message from the *condor_schedd* before killing it.

NOT_RESPONDING_WANT_CORE A boolean value with a default value of `False`. This parameter is for debugging purposes on Unix systems, and it controls the behavior of the parent process when the parent process determines that a child process is not responding. If `NOT_RESPONDING_WANT_CORE` is `True`, the parent will send a SIGABRT instead of

SIGKILL to the child process. If the child process is configured with the configuration variable CREATE_CORE_FILES enabled, the child process will then generate a core dump. See NOT_RESPONDING_TIMEOUT on page 178, and CREATE_CORE_FILES on page 168 for related details.

LOCK_FILE_UPDATE_INTERVAL An integer value representing seconds, controlling how often valid lock files should have their on disk timestamps updated. Updating the timestamps prevents administrative programs, such as *tmpwatch*, from deleting long lived lock files. If set to a value less than 60, the update time will be 60 seconds. The default value is 28800, which is 8 hours. This variable only takes effect at the start or restart of a daemon.

MAX_ACCEPTS_PER_CYCLE An integer value that defaults to 4. It is a limit on the number of accepts of new, incoming, socket connect requests per DaemonCore event cycle. It has the most noticeable effect on the *condor_schedd*, and would be given a higher integer value for tuning purposes when there is a high number of jobs starting and exiting per second.

3.3.6 Network-Related Configuration File Entries

More information about networking in Condor can be found in section 3.7 on page 358.

BIND_ALL_INTERFACES For systems with multiple network interfaces, if this configuration setting is False, Condor will only bind network sockets to the IP address specified with NETWORK_INTERFACE (described below). If set to True, the default value, Condor will listen on all interfaces. However, currently Condor is still only able to advertise a single IP address, even if it is listening on multiple interfaces. By default, it will advertise the IP address of the network interface used to contact the collector, since this is the most likely to be accessible to other processes which query information from the same collector. More information about using this setting can be found in section 3.7.3 on page 364.

CCB_ADDRESS This is the address of a *condor_collector* that will serve as this daemon's Condor Connection Broker (CCB). Multiple addresses may be listed (separated by commas and/or spaces) for redundancy. The CCB server must authorize this daemon at DAEMON level for this configuration to succeed. It is highly recommended to also configure PRIVATE_NETWORK_NAME if you configure CCB_ADDRESS so communications originating within the same private network do not need to go through CCB. For more information about CCB, see page 367.

CCB_HEARTBEAT_INTERVAL This is the maximum number of seconds of silence on a daemon's connection to the CCB server after which it will ping the server to verify that the connection still works. The default is 20 minutes. This feature serves to both speed up detection of dead connections and to generate a guaranteed minimum frequency of activity to attempt to prevent the connection from being dropped. The special value 0 disables the heartbeat. The heartbeat is automatically disabled if the CCB server is older than 7.5.0.

USE_SHARED_PORT A boolean value that specifies whether a Condor process should rely on *condor_shared_port* for receiving incoming connections. Under Unix, write access to the

location defined by DAEMON_SOCKET_DIR is required for this to take effect. The default is False. If set to True, SHARED_PORT should be added to DAEMON_LIST. For more information about using a shared port, see page 269.

<SUBSYS>_MAX_FILE_DESCRIPTORS This setting is identical to MAX_FILE_DESCRIPTORS, but it only applies to a specific condor subsystem. If the subsystem-specific setting is unspecified, MAX_FILE_DESCRIPTORS is used.

MAX_FILE_DESCRIPTORS Under Unix, this specifies the maximum number of file descriptors to allow the Condor daemon to use. File descriptors are a system resource used for open files and for network connections. Condor daemons that make many simultaneous network connections may require an increased number of file descriptors. For example, see page 367 for information on file descriptor requirements of CCB. Changes to this configuration variable require a restart of Condor in order to take effect. Also note that only if Condor is running as root will it be able to increase the limit above the hard limit (on maximum open files) that it inherits.

NETWORK_INTERFACE An IP address of the form 123.123.123.123 or the name of a network device, as in the example eth0. The wild card character (@) may be used within either. For example, 123.123.* would match a network interface with an IP address of 123.123.123.123 or 123.123.100.100. The default value is *, which matches all network interfaces.

The effect of this variable depends on the value of BIND_ALL_INTERFACES. There are two cases:

If BIND_ALL_INTERFACES is True (the default), NETWORK_INTERFACE controls what IP address will be advertised as the public address of the daemon. If multiple network interfaces match the value and ENABLE_ADDRESS_REWRITING is True (the default), the IP address that is chosen to be advertised will be the one that is used to communicate with the *condor_collector*. If ENABLE_ADDRESS_REWRITING is False, the IP address that is chosen to be advertised will be the one associated with the first device (in system-defined order) that is in a public address space, or a private address space, or a loopback address, in that order of preference. If it is desired to advertise an IP address that is not associated with any local network interface, for example, when TCP forwarding is being used, then TCP_FORWARDING_HOST should be used instead of NETWORK_INTERFACE.

If BIND_ALL_INTERFACES is False, then NETWORK_INTERFACE specifies which IP address Condor should use for all incoming and outgoing communication. If more than one IP address matches the value, then the IP address that is chosen will be the one associated with the first device (in system-defined order) that is in a public address space, or a private address space, or a loopback address, in that order of preference.

More information about configuring Condor on machines with multiple network interfaces can be found in section 3.7.3 on page 364.

PRIVATE_NETWORK_NAME If two Condor daemons are trying to communicate with each other, and they both belong to the same private network, this setting will allow them to communicate directly using the private network interface, instead of having to use CCB or the Generic Connection Broker (GCB) or to go through a public IP address. Each private network should

be assigned a unique network name. This string can have any form, but it must be unique for a particular private network. If another Condor daemon or tool is configured with the same PRIVATE_NETWORK_NAME, it will attempt to contact this daemon using its private network address. Even for sites using CCB or GCB, this is an important optimization, since it means that two daemons on the same network can communicate directly, without having to go through the broker. If CCB/GCB is enabled, and the PRIVATE_NETWORK_NAME is defined, the daemon's private address will be defined automatically. Otherwise, you can specify a particular private IP address to use by defining the PRIVATE_NETWORK_INTERFACE setting (described below). There is no default for this setting. After changing this setting and running *condor_reconfig*, it may take up to one *condor_collector* update interval before the change becomes visible.

PRIVATE_NETWORK_INTERFACE For systems with multiple network interfaces, if this configuration setting and PRIVATE_NETWORK_NAME are both defined, Condor daemons will advertise some additional attributes in their ClassAds to help other Condor daemons and tools in the same private network to communicate directly.

PRIVATE_NETWORK_INTERFACE defines what IP address of the form 123.123.123.123 or name of a network device (as in the example eth0) a given multi-homed machine should use for the private network. The asterisk (*) may be used as a wild card character within either the IP address or the device name. If another Condor daemon or tool is configured with the same PRIVATE_NETWORK_NAME, it will attempt to contact this daemon using the IP address specified here. The syntax for specifying an IP address is identical to NETWORK_INTERFACE. Sites using CCB or the Generic Connection Broker (GCB) only need to define the PRIVATE_NETWORK_NAME, and the PRIVATE_NETWORK_INTERFACE will be defined automatically. Unless CCB/GCB is enabled, there is no default value for this variable. After changing this variable and running *condor_reconfig*, it may take up to one *condor_collector* update interval before the change becomes visible.

TCP_FORWARDING_HOST This specifies the host or IP address that should be used as the public address of this daemon. If a host name is specified, be aware that it will be resolved to an IP address by this daemon, not by the clients wishing to connect to it. It is the IP address that is advertised, not the host name. This setting is useful if Condor on this host may be reached through a NAT or firewall by connecting to an IP address that forwards connections to this host. It is assumed that the port number on the TCP_FORWARDING_HOST that forwards to this host is the same port number assigned to Condor on this host. This option could also be used when ssh port forwarding is being used. In this case, the incoming addresses of connections to this daemon will appear as though they are coming from the forwarding host rather than from the real remote host, so any authorization settings that rely on host addresses should be considered accordingly.

ENABLE_ADDRESS_REWRITING A boolean value that defaults to True. When NETWORK_INTERFACE matches only one IP address or TCP_FORWARDING_HOST is defined or NET_REMAP_ENABLE is True, this setting has no effect and the behavior is as though it had been set to False. When True, IP addresses published by Condor daemons are automatically rewritten to match the IP address of the network interface used to make the publication. For example, if the *condor_schedd* advertises itself to two pools via flocking,

and the *condor_collector* for one pool is reached by the *condor_schedd* through a private network interface, while the *condor_collector* for the other pool is reached through a different network interface, the IP address published by the *condor_schedd* daemon will match the address of the respective network interfaces used in the two cases. The intention is to make it easier for Condor daemons to operate in a multi-homed environment.

HIGHPORT Specifies an upper limit of given port numbers for Condor to use, such that Condor is restricted to a range of port numbers. If this macro is not explicitly specified, then Condor will not restrict the port numbers that it uses. Condor will use system-assigned port numbers. For this macro to work, both HIGHPORT and LOWPORT (given below) must be defined.

LOWPORT Specifies a lower limit of given port numbers for Condor to use, such that Condor is restricted to a range of port numbers. If this macro is not explicitly specified, then Condor will not restrict the port numbers that it uses. Condor will use system-assigned port numbers. For this macro to work, both HIGHPORT (given above) and LOWPORT must be defined.

IN_LOWPORT An integer value that specifies a lower limit of given port numbers for Condor to use on incoming connections (ports for listening), such that Condor is restricted to a range of port numbers. This range implies the use of both IN_LOWPORT and IN_HIGHPORT. A range of port numbers less than 1024 may be used for daemons running as root. Do not specify IN_LOWPORT in combination with IN_HIGHPORT such that the range crosses the port 1024 boundary. Applies only to Unix machine configuration. Use of IN_LOWPORT and IN_HIGHPORT overrides any definition of LOWPORT and HIGHPORT.

IN_HIGHPORT An integer value that specifies an upper limit of given port numbers for Condor to use on incoming connections (ports for listening), such that Condor is restricted to a range of port numbers. This range implies the use of both IN_LOWPORT and IN_HIGHPORT. A range of port numbers less than 1024 may be used for daemons running as root. Do not specify IN_LOWPORT in combination with IN_HIGHPORT such that the range crosses the port 1024 boundary. Applies only to Unix machine configuration. Use of IN_LOWPORT and IN_HIGHPORT overrides any definition of LOWPORT and HIGHPORT.

OUT_LOWPORT An integer value that specifies a lower limit of given port numbers for Condor to use on outgoing connections, such that Condor is restricted to a range of port numbers. This range implies the use of both OUT_LOWPORT and OUT_HIGHPORT. A range of port numbers less than 1024 is inappropriate, as not all daemons and tools will be run as root. Applies only to Unix machine configuration. Use of OUT_LOWPORT and OUT_HIGHPORT overrides any definition of LOWPORT and HIGHPORT.

OUT_HIGHPORT An integer value that specifies an upper limit of given port numbers for Condor to use on outgoing connections, such that Condor is restricted to a range of port numbers. This range implies the use of both OUT_LOWPORT and OUT_HIGHPORT. A range of port numbers less than 1024 is inappropriate, as not all daemons and tools will be run as root. Applies only to Unix machine configuration. Use of OUT_LOWPORT and OUT_HIGHPORT overrides any definition of LOWPORT and HIGHPORT.

UPDATE_COLLECTOR_WITH_TCP If your site needs to use TCP connections to send ClassAd updates to your collector, set to True to enable this feature. Please read section 3.7.6 on "Using TCP to Send Collector Updates" on page 383 for more details and a discussion of when

this functionality is needed. At this time, this setting only affects the main *condor_collector* for the site, not any sites that a *condor_schedd* might flock to. For large pools, it is also necessary to ensure that the collector has a high enough file descriptor limit (e.g. using `MAX_FILE_DESCRIPTORS`. Defaults to `False`.

TCP_UPDATE_COLLECTORS The list of collectors which will be updated with TCP instead of UDP. Please read section 3.7.6 on "Using TCP to Send Collector Updates" on page 383 for more details and a discussion of when a site needs this functionality. If not defined, no collectors use TCP instead of UDP.

<SUBSYS>_TIMEOUT_MULTIPLIER An integer value that defaults to 1. This value multiplies configured timeout values for all targeted subsystem communications, thereby increasing the time until a timeout occurs. This configuration variable is intended for use by developers for debugging purposes, where communication timeouts interfere.

NONBLOCKING_COLLECTOR_UPDATE A boolean value that defaults to `True`. When `True`, the establishment of TCP connections to the *condor_collector* daemon for a security-enabled pool are done in a nonblocking manner.

NEGOTIATOR_USE_NONBLOCKING_STARTD_CONTACT A boolean value that defaults to `True`. When `True`, the establishment of TCP connections from the *condor_negotiator* daemon to the *condor_startd* daemon for a security-enabled pool are done in a nonblocking manner.

The following settings are specific to enabling Generic Connection Brokering or GCB in your Condor pool. The functionality of GCB is being replaced with CCB, so consider using CCB if it fits your needs. More information about GCB and how to configure it can be found in section 3.7.5 on page 370.

NET_REMAP_ENABLE A boolean variable, that when defined to `True`, enables a network remapping service for Condor. The service to use is controlled by `NET_REMAP_SERVICE`. This boolean value defaults to `False`.

NET_REMAP_SERVICE If `NET_REMAP_ENABLE` is defined to `True`, this setting controls what network remapping service should be used. Currently, the only value supported is `GCB`. The default is undefined.

NET_REMAP_INAGENT A comma or space-separated list of IP addresses for GCB brokers. Upon start up, the *condor_master* chooses one at random from among the working brokers in the list. There is no default if not defined.

NET_REMAP_ROUTE Hosts with the GCB network remapping service enabled that would like to use a GCB routing table GCB broker specify the full path to their routing table with this setting. There is no default value if undefined.

MASTER_WAITS_FOR_GCB_BROKER A boolean value that defaults to `True`. This variable determines the behavior of the *condor_master* with GCB enabled. With no GCB broker working upon either the start up of the *condor_master*, or once the *condor_master*

has successfully communicated with a GCB broker, but the communication fails, if MASTER_WAITS_FOR_GCB_BROKER is True, the *condor_master* waits while attempting to find a working GCB broker. With no GCB broker working upon the start up of the *condor_master*, if MASTER_WAITS_FOR_GCB_BROKER is False, the *condor_master* fails and exits, without restarting. Once the *condor_master* has successfully communicated with a GCB broker, but the communication fails, if MASTER_WAITS_FOR_GCB_BROKER is False, the *condor_master* kills all its children, exits, and restarts.

The set up task of *condor_glidein* explicitly sets MASTER_WAITS_FOR_GCB_BROKER to False in the configuration file it produces.

3.3.7 Shared File System Configuration File Macros

These macros control how Condor interacts with various shared and network file systems. If you are using AFS as your shared file system, be sure to read section 3.13.1 on Using Condor with AFS. For information on submitting jobs under shared file systems, see section 2.5.3.

UID_DOMAIN The UID_DOMAIN macro is used to decide under which user to run jobs. If the $(UID_DOMAIN) on the submitting machine is different than the $(UID_DOMAIN) on the machine that runs a job, then Condor runs the job as the user nobody. For example, if the submit machine has a $(UID_DOMAIN) of flippy.cs.wisc.edu, and the machine where the job will execute has a $(UID_DOMAIN) of cs.wisc.edu, the job will run as user nobody, because the two $(UID_DOMAIN)s are not the same. If the $(UID_DOMAIN) is the same on both the submit and execute machines, then Condor will run the job as the user that submitted the job.

A further check attempts to assure that the submitting machine can not lie about its UID_DOMAIN. Condor compares the submit machine's claimed value for UID_DOMAIN to its fully qualified name. If the two do not end the same, then the submit machine is presumed to be lying about its UID_DOMAIN. In this case, Condor will run the job as user nobody. For example, a job submission to the Condor pool at the UW Madison from flippy.example.com, claiming a UID_DOMAIN of of cs.wisc.edu, will run the job as the user nobody.

Because of this verification, $(UID_DOMAIN) must be a real domain name. At the Computer Sciences department at the UW Madison, we set the $(UID_DOMAIN) to be cs.wisc.edu to indicate that whenever someone submits from a department machine, we will run the job as the user who submits it.

Also see SOFT_UID_DOMAIN below for information about one more check that Condor performs before running a job as a given user.

A few details:

An administrator could set UID_DOMAIN to *. This will match all domains, but it is a gaping security hole. It is not recommended.

An administrator can also leave UID_DOMAIN undefined. This will force Condor to always run jobs as user nobody. Running standard universe jobs as user nobody enhances security and should cause no problems, because the jobs use remote I/O to access all of their files.

However, if vanilla jobs are run as user nobody, then files that need to be accessed by the job will need to be marked as world readable/writable so the user nobody can access them.

When Condor sends e-mail about a job, Condor sends the e-mail to user@$(UID_DOMAIN). If UID_DOMAIN is undefined, the e-mail is sent to user@submitmachinename.

TRUST_UID_DOMAIN As an added security precaution when Condor is about to spawn a job, it ensures that the UID_DOMAIN of a given submit machine is a substring of that machine's fully-qualified host name. However, at some sites, there may be multiple UID spaces that do not clearly correspond to Internet domain names. In these cases, administrators may wish to use names to describe the UID domains which are not substrings of the host names of the machines. For this to work, Condor must not do this regular security check. If the TRUST_UID_DOMAIN setting is defined to True, Condor will not perform this test, and will trust whatever UID_DOMAIN is presented by the submit machine when trying to spawn a job, instead of making sure the submit machine's host name matches the UID_DOMAIN. When not defined, the default is False, since it is more secure to perform this test.

SOFT_UID_DOMAIN A boolean variable that defaults to False when not defined. When Condor is about to run a job as a particular user (instead of as user nobody), it verifies that the UID given for the user is in the password file and actually matches the given user name. However, under installations that do not have every user in every machine's password file, this check will fail and the execution attempt will be aborted. To cause Condor not to do this check, set this configuration variable to True. Condor will then run the job under the user's UID.

SLOT<N>_USER The name of a user for Condor to use instead of user nobody, as part of a solution that plugs a security hole whereby a lurker process can prey on a subsequent job run as user name nobody. <N> is an integer associated with slots. On Windows, SLOT<N>_USER will only work if the credential of the specified user is stored on the execute machine using *condor_store_cred*. See Section 3.6.13 for more information.

STARTER_ALLOW_RUNAS_OWNER A boolean expression evaluated with the job ad as the target, that determines whether the job may run under the job owner's account (True) or whether it will run as SLOT<N>_USER or nobody (False). On Unix, this defaults to True. On Windows, it defaults to False. The job ClassAd may also contain the attribute RunAsOwner which is logically ANDed with the *condor_starter* daemon's boolean value. Under Unix, if the job does not specify it, this attribute defaults to True. Under Windows, the attribute defaults to False. In Unix, if the UidDomain of the machine and job do not match, then there is no possibility to run the job as the owner anyway, so, in that case, this setting has no effect. See Section 3.6.13 for more information.

DEDICATED_EXECUTE_ACCOUNT_REGEXP This is a regular expression (i.e. a string matching pattern) that matches the account name(s) that are dedicated to running condor jobs on the execute machine and which will never be used for more than one job at a time. The default matches no account name. If you have configured SLOT<N>_USER to be a *different* account for each Condor slot, and no non-condor processes will ever be run by these accounts, then this pattern should match the names of all SLOT<N>_USER accounts. Jobs run under a dedicated execute account are reliably tracked by Condor, whereas other jobs, may spawn processes

that Condor fails to detect. Therefore, a dedicated execution account provides more reliable tracking of CPU usage by the job and it also guarantees that when the job exits, no "lurker" processes are left behind. When the job exits, condor will attempt to kill all processes owned by the dedicated execution account. Example:

```
SLOT1_USER = cndrusr1
SLOT2_USER = cndrusr2
STARTER_ALLOW_RUNAS_OWNER = False
DEDICATED_EXECUTE_ACCOUNT_REGEXP = cndrusr[0-9]+
```

You can tell if the starter is in fact treating the account as a dedicated account, because it will print a line such as the following in its log file:

```
Tracking process family by login "cndrusr1"
```

EXECUTE_LOGIN_IS_DEDICATED This configuration setting is deprecated because it cannot handle the case where some jobs run as dedicated accounts and some do not. Use `DEDICATED_EXECUTE_ACCOUNT_REGEXP` instead.

A boolean value that defaults to `False`. When `True`, Condor knows that all jobs are being run by dedicated execution accounts (whether they are running as the job owner or as nobody or as `SLOT<N>_USER`). Therefore, when the job exits, all processes running under the same account will be killed.

FILESYSTEM_DOMAIN The `FILESYSTEM_DOMAIN` macro is an arbitrary string that is used to decide if two machines (a submitting machine and an execute machine) share a file system. Although the macro name contains the word "DOMAIN", the macro is not required to be a domain name. It often is a domain name.

Note that this implementation is not ideal: machines may share some file systems but not others. Condor currently has no way to express this automatically. You can express the need to use a particular file system by adding additional attributes to your machines and submit files, similar to the example given in Frequently Asked Questions, section 7 on how to run jobs only on machines that have certain software packages.

Note that if you do not set `$(FILESYSTEM_DOMAIN)`, Condor defaults to setting the macro's value to be the fully qualified host name of the local machine. Since each machine will have a different `$(FILESYSTEM_DOMAIN)`, they will not be considered to have shared file systems.

RESERVE_AFS_CACHE If your machine is running AFS and the AFS cache lives on the same partition as the other Condor directories, and you want Condor to reserve the space that your AFS cache is configured to use, set this macro to `True`. It defaults to `False`.

USE_NFS This macro influences how Condor jobs running in the standard universe access their files. Condor will redirect the file I/O requests of standard universe jobs to be executed on the machine which submitted the job. Because of this, as a Condor job migrates around the network, the file system always appears to be identical to the file system where the job was

submitted. However, consider the case where a user's data files are sitting on an NFS server. The machine running the user's program will send all I/O over the network to the machine which submitted the job, which in turn sends all the I/O over the network a second time back to the NFS file server. Thus, all of the program's I/O is being sent over the network twice.

If this macro to `True`, then Condor will attempt to read/write files without redirecting I/O back to the submitting machine if both the submitting machine and the machine running the job are both accessing the same NFS servers (*if* they are both in the same `$(FILESYSTEM_DOMAIN)` and in the same `$(UID_DOMAIN)`, as described above). The result is I/O performed by Condor standard universe jobs is only sent over the network once. While sending all file operations over the network twice might sound really bad, unless you are operating over networks where bandwidth as at a very high premium, practical experience reveals that this scheme offers very little real performance gain. There are also some (fairly rare) situations where this scheme can break down.

Setting `$(USE_NFS)` to `False` is always safe. It may result in slightly more network traffic, but Condor jobs are most often heavy on CPU and light on I/O. It also ensures that a remote standard universe Condor job will always use Condor's remote system calls mechanism to reroute I/O and therefore see the exact same file system that the user sees on the machine where she/he submitted the job.

Some gritty details for folks who want to know: If the you set `$(USE_NFS)` to `True`, and the `$(FILESYSTEM_DOMAIN)` of both the submitting machine and the remote machine about to execute the job match, and the `$(FILESYSTEM_DOMAIN)` claimed by the submit machine is indeed found to be a subset of what an inverse look up to a DNS (domain name server) reports as the fully qualified domain name for the submit machine's IP address (this security measure safeguards against the submit machine from lying), *then* the job will access files using a local system call, without redirecting them to the submitting machine (with NFS). Otherwise, the system call will get routed back to the submitting machine using Condor's remote system call mechanism. NOTE: When submitting a vanilla job, *condor_submit* will, by default, append requirements to the Job ClassAd that specify the machine to run the job must be in the same `$(FILESYSTEM_DOMAIN)` and the same `$(UID_DOMAIN)`.

`IGNORE_NFS_LOCK_ERRORS` When set to `True`, all errors related to file locking errors from NFS are ignored. Defaults to `False`, not ignoring errors.

`USE_AFS` If your machines have AFS, this macro determines whether Condor will use remote system calls for standard universe jobs to send I/O requests to the submit machine, or if it should use local file access on the execute machine (which will then use AFS to get to the submitter's files). Read the setting above on `$(USE_NFS)` for a discussion of why you might want to use AFS access instead of remote system calls.

One important difference between `$(USE_NFS)` and `$(USE_AFS)` is the AFS cache. With `$(USE_AFS)` set to `True`, the remote Condor job executing on some machine will start modifying the AFS cache, possibly evicting the machine owner's files from the cache to make room for its own. Generally speaking, since we try to minimize the impact of having a Condor job run on a given machine, we do not recommend using this setting.

While sending all file operations over the network twice might sound really bad, unless you are operating over networks where bandwidth as at a very high premium, practical experience

reveals that this scheme offers very little real performance gain. There are also some (fairly rare) situations where this scheme can break down.

Setting `$(USE_AFS)` to `False` is always safe. It may result in slightly more network traffic, but Condor jobs are usually heavy on CPU and light on I/O. `False` ensures that a remote standard universe Condor job will always see the exact same file system that the user on sees on the machine where he/she submitted the job. Plus, it will ensure that the machine where the job executes does not have its AFS cache modified as a result of the Condor job being there.

However, things may be different at your site, which is why the setting is there.

3.3.8 Checkpoint Server Configuration File Macros

These macros control whether or not Condor uses a checkpoint server. This section describes the settings that the checkpoint server itself needs defined. See section 3.8 on Installing a Checkpoint Server for details on installing and running a checkpoint server.

CKPT_SERVER_HOST The host name of a checkpoint server.

STARTER_CHOOSES_CKPT_SERVER If this parameter is `True` or undefined on the submit machine, the checkpoint server specified by `$(CKPT_SERVER_HOST)` on the execute machine is used. If it is `False` on the submit machine, the checkpoint server specified by `$(CKPT_SERVER_HOST)` on the submit machine is used.

CKPT_SERVER_DIR The full path of the directory the checkpoint server should use to store checkpoint files. Depending on the size of the pool and the size of the jobs submitted, this directory and its subdirectories might need to store many Mbytes of data.

USE_CKPT_SERVER A boolean which determines if a given submit machine is to use a checkpoint server if one is available. If a checkpoint server is not available or the variable `USE_CKPT_SERVER` is set to `False`, checkpoints will be written to the local `$(SPOOL)` directory on the submission machine.

MAX_DISCARDED_RUN_TIME If the *condor_shadow* daemon is unable to read a checkpoint file from the checkpoint server, it keeps trying only if the job has accumulated more than this many seconds of CPU usage. Otherwise, the job is started from scratch. Defaults to 3600 (1 hour). This variable is only used if `$(USE_CKPT_SERVER)` is `True`.

CKPT_SERVER_CHECK_PARENT_INTERVAL This is the number of seconds between checks to see whether the parent of the checkpoint server (usually the *condor_master*) has died. If the parent has died, the checkpoint server shuts itself down. The default is 120 seconds. A setting of 0 disables this check.

CKPT_SERVER_INTERVAL The maximum number of seconds the checkpoint server waits for activity on network sockets before performing other tasks. The default value is 300 seconds.

CKPT_SERVER_CLASSAD_FILE A string that represents a file in the file system to which ClassAds will be written. The ClassAds denote information about stored checkpoint files, such as

owner, shadow IP address, name of the file, and size of the file. This information is also independently recorded in the `TransferLog`. The default setting is undefined, which means a checkpoint server ClassAd file will not be kept.

CKPT_SERVER_CLEAN_INTERVAL The number of seconds that must pass until the ClassAd log file as described by the `CKPT_SERVER_CLASSAD_FILE` variable gets truncated. The default is 86400 seconds, which is one day.

CKPT_SERVER_REMOVE_STALE_CKPT_INTERVAL The number of seconds between attempts to discover and remove stale checkpoint files. It defaults to 86400 seconds, which is one day.

CKPT_SERVER_SOCKET_BUFSIZE The number of bytes representing the size of the TCP send/recv buffer on the socket file descriptor related to moving the checkpoint file to and from the checkpoint server. The default value is 0, which allows the operating system to decide the size.

CKPT_SERVER_MAX_PROCESSES The maximum number of child processes that could be working on behalf of the checkpoint server. This includes store processes and restore processes. The default value is 50.

CKPT_SERVER_MAX_STORE_PROCESSES The maximum number of child process strictly devoted to the storage of checkpoints. The default is the value of `CKPT_SERVER_MAX_PROCESSES`.

CKPT_SERVER_MAX_RESTORE_PROCESSES The maximum number of child process strictly devoted to the restoring of checkpoints. The default is the value of `CKPT_SERVER_MAX_PROCESSES`.

CKPT_SERVER_STALE_CKPT_AGE_CUTOFF The number of seconds after which if a checkpoint file has not been accessed, it is considered stale. The default value is 5184000 seconds, which is sixty days.

3.3.9 condor_master Configuration File Macros

These macros control the *condor_master*.

DAEMON_LIST This macro determines what daemons the *condor_master* will start and keep its watchful eyes on. The list is a comma or space separated list of subsystem names (listed in section 3.3.1). For example,

```
DAEMON_LIST = MASTER, STARTD, SCHEDD
```

NOTE: This configuration variable cannot be changed by using *condor_reconfig* or by sending a SIGHUP. To change this configuration variable, restart the *condor_master* daemon by using *condor_restart*. Only then will the change take effect.

NOTE: On your central manager, your `$(DAEMON_LIST)` will be different from your regular pool, since it will include entries for the *condor_collector* and *condor_negotiator*.

DC_DAEMON_LIST A list delimited by commas and/or spaces that lists the daemons in DAEMON_LIST which use the Condor DaemonCore library. The *condor_master* must differentiate between daemons that use DaemonCore and those that do not, so it uses the appropriate inter-process communication mechanisms. This list currently includes all Condor daemons except the checkpoint server by default.

As of Condor version 7.2.1, a daemon may be appended to the default DC_DAEMON_LIST value by placing the plus character (+) before the first entry in the DC_DAEMON_LIST definition. For example:

```
DC_DAEMON_LIST = +NEW_DAEMON
```

<SUBSYS> Once you have defined which subsystems you want the *condor_master* to start, you must provide it with the full path to each of these binaries. For example:

```
MASTER          = $(SBIN)/condor_master
STARTD          = $(SBIN)/condor_startd
SCHEDD          = $(SBIN)/condor_schedd
```

These are most often defined relative to the $(SBIN) macro.

The macro is named by substituting <SUBSYS> with the appropriate subsystem string as defined in section 3.3.1.

<DaemonName>_ENVIRONMENT <DaemonName> is the name of a daemon listed in DAEMON_LIST. Defines changes to the environment that the daemon is invoked with. It should use the same syntax for specifying the environment as the environment specification in a submit description file. For example, to redefine the TMP and CONDOR_CONFIG environment variables seen by the *condor_schedd*, place the following in the configuration:

```
SCHEDD_ENVIRONMENT = "TMP=/new/value CONDOR_CONFIG=/special/config"
```

When the *condor_schedd* daemon is started by the *condor_master*, it would see the specified values of TMP and CONDOR_CONFIG.

<SUBSYS>_ARGS This macro allows the specification of additional command line arguments for any process spawned by the *condor_master*. List the desired arguments using the same syntax as the arguments specification in a *condor_submit* submit file (see page 827), with one exception: do not escape double-quotes when using the old-style syntax (this is for backward compatibility). Set the arguments for a specific daemon with this macro, and the macro will affect only that daemon. Define one of these for each daemon the *condor_master* is controlling. For example, set $(STARTD_ARGS) to specify any extra command line arguments to the *condor_startd*.

The macro is named by substituting <SUBSYS> with the appropriate subsystem string as defined in section 3.3.1.

<SUBSYS>_USERID The account name that should be used to run the SUBSYS process spawned by the *condor_master*. When not defined, the process is spawned as the same user that is

running *condor_master*. When defined, the real user id of the spawned process will be set to the specified account, so if this account is not `root`, the process will not have `root` privileges. The *condor_master* must be running as root in order to start processes as other users. Example configuration:

```
COLLECTOR_USERID = condor
NEGOTIATOR_USERID = condor
```

The above example runs the *condor_collector* and *condor_negotiator* as the `condor` user with no `root` privileges. If we specified some account other than the `condor` user, as set by the (`CONDOR_IDS`) configuration variable, then we would need to configure the log files for these daemons to be in a directory that they can write to. When using GSI security or any other security method in which the daemon credential is owned by `root`, it is also necessary to make a copy of the credential, make it be owned by the account the daemons are using, and configure the daemons to use that copy.

PREEN In addition to the daemons defined in `$(DAEMON_LIST)`, the *condor_master* also starts up a special process, *condor_preen* to clean out junk files that have been left laying around by Condor. This macro determines where the *condor_master* finds the *condor_preen* binary. If this macro is set to nothing, *condor_preen* will not run.

PREEN_ARGS Controls how *condor_preen* behaves by allowing the specification of command-line arguments. This macro works as `$(<SUBSYS>_ARGS)` does. The difference is that you must specify this macro for *condor_preen* if you want it to do anything. *condor_preen* takes action only because of command line arguments. **-m** means you want e-mail about files *condor_preen* finds that it thinks it should remove. **-r** means you want *condor_preen* to actually remove these files.

PREEN_INTERVAL This macro determines how often *condor_preen* should be started. It is defined in terms of seconds and defaults to 86400 (once a day).

PUBLISH_OBITUARIES When a daemon crashes, the *condor_master* can send e-mail to the address specified by `$(CONDOR_ADMIN)` with an obituary letting the administrator know that the daemon died, the cause of death (which signal or exit status it exited with), and (optionally) the last few entries from that daemon's log file. If you want obituaries, set this macro to `True`.

OBITUARY_LOG_LENGTH This macro controls how many lines of the log file are part of obituaries. This macro has a default value of 20 lines.

START_MASTER If this setting is defined and set to `False` when the *condor_master* starts up, the first thing it will do is exit. This appears strange, but perhaps you do not want Condor to run on certain machines in your pool, yet the boot scripts for your entire pool are handled by a centralized This is an entry you would most likely find in a local configuration file, not a global configuration file.

START_DAEMONS This macro is similar to the `$(START_MASTER)` macro described above. However, the *condor_master* does not exit; it does not start any of the daemons listed in the `$(DAEMON_LIST)`. The daemons may be started at a later time with a *condor_on* command.

MASTER_UPDATE_INTERVAL This macro determines how often the *condor_master* sends a ClassAd update to the *condor_collector*. It is defined in seconds and defaults to 300 (every 5 minutes).

MASTER_CHECK_NEW_EXEC_INTERVAL This macro controls how often the *condor_master* checks the timestamps of the running daemons. If any daemons have been modified, the master restarts them. It is defined in seconds and defaults to 300 (every 5 minutes).

MASTER_NEW_BINARY_DELAY Once the *condor_master* has discovered a new binary, this macro controls how long it waits before attempting to execute the new binary. This delay exists because the *condor_master* might notice a new binary while it is in the process of being copied, in which case trying to execute it yields unpredictable results. The entry is defined in seconds and defaults to 120 (2 minutes).

SHUTDOWN_FAST_TIMEOUT This macro determines the maximum amount of time daemons are given to perform their fast shutdown procedure before the *condor_master* kills them outright. It is defined in seconds and defaults to 300 (5 minutes).

MASTER_SHUTDOWN_<Name> A full path and file name of a program that the *condor_master* is to execute via the Unix `execl()` call, or the similar Win32 `_execl()` call, instead of the normal call to `exit()`. Multiple programs to execute may be defined with multiple entries, each with a unique `Name`. These macros have no affect on a *condor_master* unless *condor_set_shutdown* is run. The `Name` specified as an argument to the *condor_set_shutdown* program must match the `Name` portion of one of these `MASTER_SHUTDOWN_<Name>` macros; if not, the *condor_master* will log an error and ignore the command. If a match is found, the *condor_master* will attempt to verify the program, and it will store the path and program name. When the *condor_master* shuts down (that is, just before it exits), the program is then executed as described above. The manual page for *condor_set_shutdown* on page 805 contains details on the use of this program.

NOTE: This program will be run with root privileges under Unix or administrator privileges under Windows. The administrator must ensure that this cannot be used in such a way as to violate system integrity.

MASTER_BACKOFF_CONSTANT and MASTER_<name>_BACKOFF_CONSTANT When a daemon crashes, *condor_master* uses an exponential back off delay before restarting it; see the discussion at the end of this section for a detailed discussion on how these parameters work together. These settings define the constant value of the expression used to determine how long to wait before starting the daemon again (and, effectively becomes the initial backoff time). It is an integer in units of seconds, and defaults to 9 seconds.

`$(MASTER_<name>_BACKOFF_CONSTANT)` is the daemon-specific form of `MASTER_BACKOFF_CONSTANT`; if this daemon-specific macro is not defined for a specific daemon, the non-daemon-specific value will used.

MASTER_BACKOFF_FACTOR and MASTER_<name>_BACKOFF_FACTOR When a daemon crashes, *condor_master* uses an exponential back off delay before restarting it; see the discussion at the end of this section for a detailed discussion on how these parameters work together. This setting is the base of the exponent used to determine how long to wait before starting the daemon again. It defaults to 2 seconds.

`$(MASTER_<name>_BACKOFF_FACTOR)` is the daemon-specific form of `MASTER_BACKOFF_FACTOR`; if this daemon-specific macro is not defined for a specific daemon, the non-daemon-specific value will used.

`MASTER_BACKOFF_CEILING` and `MASTER_<name>_BACKOFF_CEILING` When a daemon crashes, *condor_master* uses an exponential back off delay before restarting it; see the discussion at the end of this section for a detailed discussion on how these parameters work together. This entry determines the maximum amount of time you want the master to wait between attempts to start a given daemon. (With 2.0 as the `$(MASTER_BACKOFF_FACTOR)`, 1 hour is obtained in 12 restarts). It is defined in terms of seconds and defaults to 3600 (1 hour).

`$(MASTER_<name>_BACKOFF_CEILING)` is the daemon-specific form of `MASTER_BACKOFF_CEILING`; if this daemon-specific macro is not defined for a specific daemon, the non-daemon-specific value will used.

`MASTER_RECOVER_FACTOR` and `MASTER_<name>_RECOVER_FACTOR` A macro to set how long a daemon needs to run without crashing before it is considered *recovered*. Once a daemon has recovered, the number of restarts is reset, so the exponential back off returns to its initial state. The macro is defined in terms of seconds and defaults to 300 (5 minutes).

`$(MASTER_<name>_RECOVER_FACTOR)` is the daemon-specific form of `MASTER_RECOVER_FACTOR`; if this daemon-specific macro is not defined for a specific daemon, the non-daemon-specific value will used.

When a daemon crashes, *condor_master* will restart the daemon after a delay (a back off). The length of this delay is based on how many times it has been restarted, and gets larger after each crashes. The equation for calculating this backoff time is given by:

$$t = c + k^n$$

where t is the calculated time, c is the constant defined by `$(MASTER_BACKOFF_CONSTANT)`, k is the "factor" defined by `$(MASTER_BACKOFF_FACTOR)`, and n is the number of restarts already attempted (0 for the first restart, 1 for the next, etc.).

With default values, after the first crash, the delay would be $t = 9 + 2.0^0$, giving 10 seconds (remember, $n = 0$). If the daemon keeps crashing, the delay increases.

For example, take the `$(MASTER_BACKOFF_FACTOR)` (which defaults to 2.0) to the power the number of times the daemon has restarted, and add `$(MASTER_BACKOFF_CONSTANT)` (which defaults to 9). Thus:

1^{st} crash: $n = 0$, so: $t = 9 + 2^0 = 9 + 1 = 10\ seconds$

2^{nd} crash: $n = 1$, so: $t = 9 + 2^1 = 9 + 2 = 11\ seconds$

3^{rd} crash: $n = 2$, so: $t = 9 + 2^2 = 9 + 4 = 13\ seconds$

...

6^{th} crash: $n = 5$, so: $t = 9 + 2^5 = 9 + 32 = 41\ seconds$

...

9^{th} crash: $n = 8$, so: $t = 9 + 2^8 = 9 + 256 = 265\ seconds$

And, after the 13 crashes, it would be:

13^{th} crash: $n = 12$, so: $t = 9 + 2^{12} = 9 + 4096 = 4105\ seconds$

This is bigger than the `$(MASTER_BACKOFF_CEILING)`, which defaults to 3600, so the daemon would really be restarted after only 3600 seconds, not 4105. The *condor_master* tries again every hour (since the numbers would get larger and would always be capped by the ceiling). Eventually, imagine that daemon finally started and did not crash. This might happen if, for example, an administrator reinstalled an accidentally deleted binary after receiving e-mail about the daemon crashing. If it stayed alive for `$(MASTER_RECOVER_FACTOR)` seconds (defaults to 5 minutes), the count of how many restarts this daemon has performed is reset to 0.

The moral of the example is that the defaults work quite well, and you probably will not want to change them for any reason.

MASTER_NAME Defines a unique name given for a *condor_master* daemon on a machine. For a *condor_master* running as `root`, it defaults to the fully qualified host name. When *not* running as `root`, it defaults to the user that instantiates the *condor_master*, concatenated with an at symbol (@), concatenated with the fully qualified host name. If more than one *condor_master* is running on the same host, then the `MASTER_NAME` for each *condor_master* must be defined to uniquely identify the separate daemons.

A defined `MASTER_NAME` is presumed to be of the form `identifying-string@full.host.name`. If the string does not include an `@` sign, Condor appends one, followed by the fully qualified host name of the local machine. The `identifying-string` portion may contain any alphanumeric ASCII characters or punctuation marks, except the `@` sign. We recommend that the string does not contain the `:` (colon) character, since that might cause problems with certain tools. Previous to Condor 7.1.1, when the string included an `@` sign, Condor replaced whatever followed the `@` sign with the fully qualified host name of the local machine. Condor does not modify any portion of the string, if it contains an `@` sign. This is useful for remote job submissions under the high availability of the job queue.

If the `MASTER_NAME` setting is used, and the *condor_master* is configured to spawn a *condor_schedd*, the name defined with `MASTER_NAME` takes precedence over the `SCHEDD_NAME` setting (see section 3.3.11 on page 213). Since Condor makes the assumption that there is only one instance of the *condor_startd* running on a machine, the `MASTER_NAME` is not automatically propagated to the *condor_startd*. However, in situations where multiple *condor_startd* daemons are running on the same host (for example, when using *condor_glidein*), the `STARTD_NAME` should be set to uniquely identify the *condor_startd* daemons (this is done automatically in the case of *condor_glidein*).

If a Condor daemon (master, schedd or startd) has been given a unique name, all Condor tools that need to contact that daemon can be told what name to use via the **-name** command-line option.

MASTER_ATTRS This macro is described in section 3.3.5 as `<SUBSYS>_ATTRS`.

MASTER_DEBUG This macro is described in section 3.3.4 as <SUBSYS>_DEBUG.

MASTER_ADDRESS_FILE This macro is described in section 3.3.5 as <SUBSYS>_ADDRESS_FILE.

SECONDARY_COLLECTOR_LIST This macro has been removed as of Condor version 6.9.3. Use the COLLECTOR_HOST configuration variable, which may define a list of *condor_collector* daemons.

ALLOW_ADMIN_COMMANDS If set to NO for a given host, this macro disables administrative commands, such as *condor_restart*, *condor_on*, and *condor_off*, to that host.

MASTER_INSTANCE_LOCK Defines the name of a file for the *condor_master* daemon to lock in order to prevent multiple *condor_master*s from starting. This is useful when using shared file systems like NFS which do not technically support locking in the case where the lock files reside on a local disk. If this macro is not defined, the default file name will be $(LOCK)/InstanceLock. $(LOCK) can instead be defined to specify the location of all lock files, not just the *condor_master*'s InstanceLock. If $(LOCK) is undefined, then the master log itself is locked.

ADD_WINDOWS_FIREWALL_EXCEPTION When set to False, the *condor_master* will not automatically add Condor to the Windows Firewall list of trusted applications. Such trusted applications can accept incoming connections without interference from the firewall. This only affects machines running Windows XP SP2 or higher. The default is True.

WINDOWS_FIREWALL_FAILURE_RETRY An integer value (default value is 60) that represents the number of times the *condor_master* will retry to add firewall exceptions. When a Windows machine boots up, Condor starts up by default as well. Under certain conditions, the *condor_master* may have difficulty adding exceptions to the Windows Firewall because of a delay in other services starting up. Examples of services that may possibly be slow are the SharedAccess service, the Netman service, or the Workstation service. This configuration variable allows administrators to set the number of times (once every 10 seconds) that the *condor_master* will retry to add firewall exceptions. A value of 0 means that Condor will retry indefinitely.

USE_PROCESS_GROUPS A boolean value that defaults to True. When False, Condor daemons on Unix machines will *not* create new sessions or process groups. Condor uses processes groups to help it track the descendants of processes it creates. This can cause problems when Condor is run under another job execution system (e.g. Condor Glidein).

3.3.10 condor_startd Configuration File Macros

NOTE: If you are running Condor on a multi-CPU machine, be sure to also read section 3.13.9 on page 443 which describes how to set up and configure Condor on SMP machines.

These settings control general operation of the *condor_startd*. Examples using these configuration macros, as well as further explanation is found in section 3.5 on Configuring The Startd Policy.

START A boolean expression that, when `True`, indicates that the machine is willing to start running a Condor job. `START` is considered when the *condor_negotiator* daemon is considering evicting the job to replace it with one that will generate a better rank for the *condor_startd* daemon, or a user with a higher priority.

SUSPEND A boolean expression that, when `True`, causes Condor to suspend running a Condor job. The machine may still be claimed, but the job makes no further progress, and Condor does not generate a load on the machine.

PREEMPT A boolean expression that, when `True`, causes Condor to stop a currently running job.

WANT_HOLD A boolean expression that defaults to `False`. When `True` and the value of `PREEMPT` becomes `True`, the job is put on hold for the reason (optionally) specified by the variables `WANT_HOLD_REASON` and `WANT_HOLD_SUBCODE`. As usual, the job owner may specify **periodic_release** and/or **periodic_remove** expressions to react to specific hold states automatically. The attribute `HoldReasonCode` in the job ClassAd is set to the value 21 when `WANT_HOLD` is responsible for putting the job on hold.

Here is an example policy that puts jobs on hold that use too much virtual memory:

```
VIRTUAL_MEMORY_AVAILABLE_MB = (VirtualMemory*0.9)
MEMORY_EXCEEDED = ImageSize/1024 > $(VIRTUAL_MEMORY_AVAILABLE_MB)
PREEMPT = ($(PREEMPT)) || ($(MEMORY_EXCEEDED))
WANT_SUSPEND = ($(WANT_SUSPEND)) && ($(MEMORY_EXCEEDED)) =!= TRUE
WANT_HOLD = ($(MEMORY_EXCEEDED))
WANT_HOLD_REASON = \
   ifThenElse( $(MEMORY_EXCEEDED), \
               "Your job used too much virtual memory.", \
               undefined )
```

WANT_HOLD_REASON An expression that defines a string utilized to set the job ClassAd attribute `HoldReason` when a job is put on hold due to `WANT_HOLD`. If not defined or if the expression evaluates to `Undefined`, a default hold reason is provided.

WANT_HOLD_SUBCODE An expression that defines an integer value utilized to set the job ClassAd attribute `HoldReasonSubCode` when a job is put on hold due to `WANT_HOLD`. If not defined or if the expression evaluates to `Undefined`, the value is set to 0. Note that `HoldReasonCode` is always set to 21.

CONTINUE A boolean expression that, when `True`, causes Condor to continue the execution of a suspended job.

KILL A boolean expression that, when `True`, causes Condor to immediately stop the execution of a currently running job, without delay, and without taking the time to produce a checkpoint (for a standard universe job).

PERIODIC_CHECKPOINT A boolean expression that, when `True`, causes Condor to initiate a checkpoint of the currently running job. This setting applies to all standard universe jobs and to vm universe jobs that have set **vm_checkpoint** to `True` in the submit description file.

RANK A floating point value that Condor uses to compare potential jobs. A larger value for a specific job ranks that job above others with lower values for RANK.

IS_VALID_CHECKPOINT_PLATFORM A boolean expression that is logically ANDed with the with the START expression to limit which machines a standard universe job may continue execution on once they have produced a checkpoint. The default expression is

```
IS_VALID_CHECKPOINT_PLATFORM =
(
  ( (TARGET.JobUniverse == 1) == FALSE) ||

  (
    (MY.CheckpointPlatform =!= UNDEFINED) &&
    (
      (TARGET.LastCheckpointPlatform =?= MY.CheckpointPlatform) ||
      (TARGET.NumCkpts == 0)
    )
  )
)
```

WANT_SUSPEND A boolean expression that, when True, tells Condor to evaluate the SUSPEND expression to decide whether to suspend a running job. When not explicitly set, the *condor_startd* exits with an error. When explicitly set, but the evaluated value is anything other than True, the value is utilized as if it were False.

WANT_VACATE A boolean expression that, when True, defines that a preempted Condor job is to be vacated, instead of killed.

IS_OWNER A boolean expression that defaults to being defined as

```
IS_OWNER = (START =?= FALSE)
```

Used to describe the state of the machine with respect to its use by its owner. Job ClassAd attributes are not used in defining IS_OWNER, as they would be Undefined.

STARTD_HISTORY A file name where the *condor_startd* daemon will maintain a job history file in an analogous way to that of the history file defined by the configuration variable HISTORY. It will be rotated in the same way, and the same parameters that apply to the HISTORY file rotation apply to the *condor_startd* daemon history as well.

STARTER This macro holds the full path to the *condor_starter* binary that the *condor_startd* should spawn. It is normally defined relative to $(SBIN).

KILLING_TIMEOUT The amount of time in seconds that the *condor_startd* should wait after sending a job-defined signal and before forcibly killing the job. Applies to all job universes other than the standard universe. The default value is 30 seconds.

POLLING_INTERVAL When a *condor_startd* enters the claimed state, this macro determines how often the state of the machine is polled to check the need to suspend, resume, vacate or kill the job. It is defined in terms of seconds and defaults to 5.

UPDATE_INTERVAL Determines how often the *condor_startd* should send a ClassAd update to the *condor_collector*. The *condor_startd* also sends update on any state or activity change, or if the value of its START expression changes. See section 3.5.5 on *condor_startd* states, section 3.5.6 on *condor_startd* Activities, and section 3.5.2 on *condor_startd* START expression for details on states, activities, and the START expression. This macro is defined in terms of seconds and defaults to 300 (5 minutes).

UPDATE_OFFSET An integer value representing the number of seconds of delay that the *condor_startd* should wait before sending its initial update, and the first update after a *condor_reconfig* command is sent to the *condor_collector*. The time of all other updates sent after this initial update is determined by $(UPDATE_INTERVAL). Thus, the first update will be sent after $(UPDATE_OFFSET) seconds, and the second update will be sent after $(UPDATE_OFFSET) + $(UPDATE_INTERVAL). This is useful when used in conjunction with the $RANDOM_INTEGER() macro for large pools, to spread out the updates sent by a large number of *condor_startd* daemons. Defaults to zero. The example configuration

```
startd.UPDATE_INTERVAL = 300
startd.UPDATE_OFFSET   = $RANDOM_INTEGER(0,300)
```

causes the initial update to occur at a random number of seconds falling between 0 and 300, with all further updates occurring at fixed 300 second intervals following the initial update.

MAXJOBRETIREMENTTIME An integer value representing the number of seconds a preempted job will be allowed to run before being evicted. The default value of 0 (when the configuration variable is not present) means that the job gets no retirement time. Note that in peaceful shutdown mode of the *condor_startd*, retirement time is treated as though infinite. In graceful shutdown mode, the job will not be preempted until the configured retirement time expires or SHUTDOWN_GRACEFUL_TIMEOUT expires. In fast shutdown mode, retirement time is ignored. See MAXJOBRETIREMENTTIME in section 3.5.8 for further explanation.

CLAIM_WORKLIFE If provided, this expression specifies the number of seconds after which a claim will stop accepting additional jobs. By default, once the negotiator gives a schedd a claim to a slot, the schedd will keep running jobs on that slot as long as it has more jobs with matching requirements, without returning the slot to the unclaimed state and renegotiating for machines. Once CLAIM_WORKLIFE expires, any existing job may continue to run as usual, but once it finishes or is preempted, the claim is closed. This may be useful if you want to force periodic renegotiation of resources without preemption having to occur. For example, if you have some low-priority jobs which should never be interrupted with kill signals, you could prevent them from being killed with MaxJobRetirementTime, but now high-priority jobs may have to wait in line when they match to a machine that is busy running one of these uninterruptible jobs. You can prevent the high-priority jobs from ever matching to such a machine by using a rank expression in the job or in the negotiator's rank expressions, but then the low-priority claim will never be interrupted; it can keep running more jobs. The solution is to use CLAIM_WORKLIFE to force the claim to stop running additional jobs after a certain amount of time. The default value for CLAIM_WORKLIFE is -1, which is treated as an infinite

claim worklife, so claims may be held indefinitely (as long as they are not preempted and the schedd does not relinquish them, of course). A value of 0 has the effect of not allowing more than one job to run per claim, since it immediately expires after the first job starts running.

MAX_CLAIM_ALIVES_MISSED The *condor_schedd* sends periodic updates to each *condor_startd* as a keep alive (see the description of ALIVE_INTERVAL on page 211). If the *condor_startd* does not receive any keep alive messages, it assumes that something has gone wrong with the *condor_schedd* and that the resource is not being effectively used. Once this happens, the *condor_startd* considers the claim to have timed out, it releases the claim, and starts advertising itself as available for other jobs. Because these keep alive messages are sent via UDP, they are sometimes dropped by the network. Therefore, the *condor_startd* has some tolerance for missed keep alive messages, so that in case a few keep alives are lost, the *condor_startd* will not immediately release the claim. This setting controls how many keep alive messages can be missed before the *condor_startd* considers the claim no longer valid. The default is 6.

STARTD_HAS_BAD_UTMP When the *condor_startd* is computing the idle time of all the users of the machine (both local and remote), it checks the `utmp` file to find all the currently active ttys, and only checks access time of the devices associated with active logins. Unfortunately, on some systems, `utmp` is unreliable, and the *condor_startd* might miss keyboard activity by doing this. So, if your `utmp` is unreliable, set this macro to `True` and the *condor_startd* will check the access time on all tty and pty devices.

CONSOLE_DEVICES This macro allows the *condor_startd* to monitor console (keyboard and mouse) activity by checking the access times on special files in `/dev`. Activity on these files shows up as `ConsoleIdle` time in the *condor_startd*'s ClassAd. Give a comma-separated list of the names of devices considered the console, without the `/dev/` portion of the path name. The defaults vary from platform to platform, and are usually correct.

One possible exception to this is on Linux, where we use "mouse" as one of the entries. Most Linux installations put in a soft link from `/dev/mouse` that points to the appropriate device (for example, `/dev/psaux` for a PS/2 bus mouse, or `/dev/tty00` for a serial mouse connected to com1). However, if your installation does not have this soft link, you will either need to put it in (you will be glad you did), or change this macro to point to the right device.

Unfortunately, modern versions of Linux do not update the access time of device files for USB devices. Thus, these files cannot be be used to determine when the console is in use. Instead, use the *condor_kbdd* daemon, which gets this information by connecting to the X server.

STARTD_JOB_EXPRS When the machine is claimed by a remote user, the *condor_startd* can also advertise arbitrary attributes from the job ClassAd in the machine ClassAd. List the attribute names to be advertised. NOTE: Since these are already ClassAd expressions, do not do anything unusual with strings. This setting defaults to "JobUniverse".

STARTD_ATTRS This macro is described in section 3.3.5 as `<SUBSYS>_ATTRS`.

STARTD_DEBUG This macro (and other settings related to debug logging in the *condor_startd*) is described in section 3.3.4 as `<SUBSYS>_DEBUG`.

STARTD_ADDRESS_FILE This macro is described in section 3.3.5 as <SUBSYS>_ADDRESS_FILE

STARTD_SHOULD_WRITE_CLAIM_ID_FILE The *condor_startd* can be configured to write out the ClaimId for the next available claim on all slots to separate files. This boolean attribute controls whether the *condor_startd* should write these files. The default value is True.

STARTD_CLAIM_ID_FILE This macro controls what file names are used if the above STARTD_SHOULD_WRITE_CLAIM_ID_FILE is true. By default, Condor will write the ClaimId into a file in the $(LOG) directory called .startd_claim_id.slotX, where X is the value of SlotID, the integer that identifies a given slot on the system, or 1 on a single-slot machine. If you define your own value for this setting, you should provide a full path, and Condor will automatically append the .slotX portion of the file name.

SlotWeight This may be used to give a slot greater weight when calculating usage, computing fair shares, and enforcing group quotas. For example, claiming a slot with SlotWeight = 2 is equivalent to claiming two SlotWeight = 1 slots. The default value is Cpus, the number of CPUs associated with the slot, which is 1 unless specially configured. Any expression referring to attributes of the slot ClassAd and evaluating to a positive floating point number is valid.

NUM_CPUS An integer value, which can be used to lie to the *condor_startd* daemon about how many CPUs a machine has. When set, it overrides the value determined with Condor's automatic computation of the number of CPUs in the machine. Lying in this way can allow multiple Condor jobs to run on a single-CPU machine, by having that machine treated like an SMP machine with multiple CPUs, which could have different Condor jobs running on each one. Or, an SMP machine may advertise more slots than it has CPUs. However, lying in this manner will hurt the performance of the jobs, since now multiple jobs will run on the same CPU, and the jobs will compete with each other. The option is only meant for people who specifically want this behavior and know what they are doing. It is disabled by default.

The default value is equal to DETECTED_CORES minus hyperthreaded cores if COUNT_HYPERTHREAD_CPUS is false. If that value exceeds MAX_NUM_CPUS , then the latter is used instead.

Note that this setting cannot be changed with a simple reconfigure, either by sending a SIGHUP or by using the *condor_reconfig* command. To change this, restart the *condor_startd* daemon for the change to take effect. The command will be

```
condor_restart -startd
```

If lying about a given machine, this fact should probably be advertised in the machine's ClassAd by using the STARTD_ATTRS setting. This way, jobs submitted in the pool could specify that they did or did not want to be matched with machines that were only really offering these fractional CPUs.

MAX_NUM_CPUS An integer value used as a ceiling for the number of CPUs detected by Condor on a machine. This value is ignored if NUM_CPUS is set. If set to zero, there is no ceiling. If not defined, the default value is zero, and thus there is no ceiling.

Note that this setting cannot be changed with a simple reconfigure, either by sending a SIGHUP or by using the *condor_reconfig* command. To change this, restart the *condor_startd* daemon for the change to take effect. The command will be

```
condor_restart -startd
```

COUNT_HYPERTHREAD_CPUS This macro controls how Condor sees hyper threaded processors. When set to `True` (the default), it includes virtual CPUs in the default value of `NUM_CPUS`. On dedicated cluster nodes, counting virtual CPUs can sometimes improve total throughput at the expense of individual job speed. However, counting them on desktop workstations can interfere with interactive job performance.

MEMORY Normally, Condor will automatically detect the amount of physical memory available on your machine. Define `MEMORY` to tell Condor how much physical memory (in MB) your machine has, overriding the value Condor computes automatically. The actual amount of memory detected by Condor is always available in the pre-defined configuration macro `DETECTED_MEMORY`.

RESERVED_MEMORY How much memory would you like reserved from Condor? By default, Condor considers all the physical memory of your machine as available to be used by Condor jobs. If `RESERVED_MEMORY` is defined, Condor subtracts it from the amount of memory it advertises as available.

STARTD_NAME Used to give an alternative value to the `Name` attribute in the *condor_startd*'s ClassAd. This esoteric configuration macro might be used in the situation where there are two *condor_startd* daemons running on one machine, and each reports to the same *condor_collector*. Different names will distinguish the two daemons. See the description of `MASTER_NAME` in section 3.3.9 on page 194 for defaults and composition of valid Condor daemon names.

RUNBENCHMARKS Specifies when to run benchmarks. When the machine is in the Unclaimed state and this expression evaluates to `True`, benchmarks will be run. If RunBenchmarks is specified and set to anything other than `False`, additional benchmarks will be run when the *condor_startd* initially starts. To disable start up benchmarks, set `RunBenchmarks` to `False`, or comment it out of the configuration file.

DedicatedScheduler A string that identifies the dedicated scheduler this machine is managed by. Section 3.13.10 on page 453 details the use of a dedicated scheduler.

STARTD_NOCLAIM_SHUTDOWN The number of seconds to run without receiving a claim before shutting Condor down on this machine. Defaults to unset, which means to never shut down. This is primarily intended for condor_glidein. Use in other situations is not recommended.

These macros control if the *condor_startd* daemon should perform backfill computations whenever resources would otherwise be idle. See section 3.13.11 on page 456 on Configuring Condor for Running Backfill Jobs for details.

ENABLE_BACKFILL A boolean value that, when `True`, indicates that the machine is willing to perform backfill computations when it would otherwise be idle. This is not a policy expression that is evaluated, it is a simple `True` or `False`. This setting controls if any of the other backfill-related expressions should be evaluated. The default is `False`.

BACKFILL_SYSTEM A string that defines what backfill system to use for spawning and managing backfill computations. Currently, the only supported value for this is `"BOINC"`, which stands for the *Berkeley Open Infrastructure for Network Computing*. See http://boinc.berkeley.edu for more information about BOINC. There is no default value, administrators must define this.

START_BACKFILL A boolean expression that is evaluated whenever a Condor resource is in the Unclaimed/Idle state and the `ENABLE_BACKFILL` expression is `True`. If `START_BACKFILL` evaluates to `True`, the machine will enter the Backfill state and attempt to spawn a backfill computation. This expression is analogous to the `START` expression that controls when a Condor resource is available to run normal Condor jobs. The default value is `False` (which means do not spawn a backfill job even if the machine is idle and `ENABLE_BACKFILL` expression is `True`). For more information about policy expressions and the Backfill state, see section 3.5 beginning on page 284, especially sections 3.5.5, 3.5.6, and 3.5.7.

EVICT_BACKFILL A boolean expression that is evaluated whenever a Condor resource is in the Backfill state which, when `True`, indicates the machine should immediately kill the currently running backfill computation and return to the Owner state. This expression is a way for administrators to define a policy where interactive users on a machine will cause backfill jobs to be removed. The default value is `False`. For more information about policy expressions and the Backfill state, see section 3.5 beginning on page 284, especially sections 3.5.5, 3.5.6, and 3.5.7.

These macros only apply to the *condor_startd* daemon when it is running on an SMP machine. See section 3.13.9 on page 443 on Configuring The Startd for SMP Machines for details.

STARTD_RESOURCE_PREFIX A string which specifies what prefix to give the unique Condor resources that are advertised on SMP machines. Previously, Condor used the term *virtual machine* to describe these resources, so the default value for this setting was “vm”. However, to avoid confusion with other kinds of virtual machines (the ones created using tools like VMware or Xen), the old *virtual machine* terminology has been changed, and we now use the term *slot*. Therefore, the default value of this prefix is now “slot”. If sites want to keep using “vm”, or prefer something other “slot”, this setting enables sites to define what string the *condor_startd* will use to name the individual resources on an SMP machine.

SLOTS_CONNECTED_TO_CONSOLE An integer which indicates how many of the machine slots the *condor_startd* is representing should be "connected" to the console (in other words, notice when there’s console activity). This defaults to all slots (N in a machine with N CPUs).

SLOTS_CONNECTED_TO_KEYBOARD An integer which indicates how many of the machine slots the *condor_startd* is representing should be "connected" to the keyboard (for remote tty activity, as well as console activity). Defaults to 1.

DISCONNECTED_KEYBOARD_IDLE_BOOST If there are slots not connected to either the keyboard or the console, the corresponding idle time reported will be the time since the *condor_startd* was spawned, plus the value of this macro. It defaults to 1200 seconds (20 minutes). We do this because if the slot is configured not to care about keyboard activity, we want it to be available to Condor jobs as soon as the *condor_startd* starts up, instead of having to wait for 15 minutes or more (which is the default time a machine must be idle before Condor will start a job). If you do not want this boost, set the value to 0. If you change your START expression to require more than 15 minutes before a job starts, but you still want jobs to start right away on some of your SMP nodes, increase this macro's value.

STARTD_SLOT_ATTRS The list of ClassAd attribute names that should be shared across all slots on the same machine. This setting was formerly know as `STARTD_VM_ATTRS` or `STARTD_VM_EXPRS` (before version 6.9.3). For each attribute in the list, the attribute's value is taken from each slot's machine ClassAd and placed into the machine ClassAd of all the other slots within the machine. For example, if the configuration file for a 2-slot machine contains

```
STARTD_SLOT_ATTRS = State, Activity, EnteredCurrentActivity
```

then the machine ClassAd for both slots will contain attributes that will be of the form:

```
slot1_State = "Claimed"
slot1_Activity = "Busy"
slot1_EnteredCurrentActivity = 1075249233
slot2_State = "Unclaimed"
slot2_Activity = "Idle"
slot2_EnteredCurrentActivity = 1075240035
```

The following settings control the number of slots reported for a given SMP host, and what attributes each one has. They are only needed if you do not want to have an SMP machine report to Condor with a separate slot for each CPU, with all shared system resources evenly divided among them. Please read section 3.13.9 on page 444 for details on how to properly configure these settings to suit your needs.

NOTE: You can only change the number of each type of slot the *condor_startd* is reporting with a simple reconfig (such as sending a SIGHUP signal, or using the *condor_reconfig* command). You cannot change the definition of the different slot types with a reconfig. If you change them, you must restart the *condor_startd* for the change to take effect (for example, using `condor_restart -startd`).

NOTE: Prior to version 6.9.3, any settings that included the term "slot" used to use "virtual machine" or "vm". If you're looking for information about one of these older settings, search for the corresponding attribute names using "slot", instead.

MAX_SLOT_TYPES The maximum number of different slot types. Note: this is the maximum number of different *types*, not of actual slots. Defaults to 10. (You should only need to change this setting if you define more than 10 separate slot types, which would be pretty rare.)

SLOT_TYPE_<N> This setting defines a given slot type, by specifying what part of each shared system resource (like RAM, swap space, etc) this kind of slot gets. This setting has *no* effect unless you also define NUM_SLOTS_TYPE_<N>. N can be any integer from 1 to the value of $(MAX_SLOT_TYPES), such as SLOT_TYPE_1. The format of this entry can be somewhat complex, so please refer to section 3.13.9 on page 444 for details on the different possibilities.

SLOT_TYPE_<N>_PARTITIONABLE A boolean variable that defaults to False. When True, this slot permits dynamic provisioning, as specified in section 3.13.9.

NUM_SLOTS_TYPE_<N> This macro controls how many of a given slot type are actually reported to Condor. There is no default.

NUM_SLOTS An integer value representing the number of slots reported when the SMP machine is being evenly divided, and the slot type settings described above are not being used. The default is one slot for each CPU. This setting can be used to reserve some CPUs on an SMP which would not be reported to the Condor pool. This value cannot be used to make Condor advertise more slots than there are CPUs on the machine. To do that, use NUM_CPUS .

ALLOW_VM_CRUFT A boolean value that Condor sets and uses internally, currently defaulting to True. When True, Condor looks for configuration variables named with the previously used string VM after searching unsuccessfully for variables named with the currently used string SLOT. When False, Condor does *not* look for variables named with the previously used string VM after searching unsuccessfully for the string SLOT.

The following configuration variables support java universe jobs.

JAVA The full path to the Java interpreter (the Java Virtual Machine).

JAVA_CLASSPATH_ARGUMENT The command line argument to the Java interpreter (the Java Virtual Machine) that specifies the Java Classpath. Classpath is a Java-specific term that denotes the list of locations (.jar files and/or directories) where the Java interpreter can look for the Java class files that a Java program requires.

JAVA_CLASSPATH_SEPARATOR The single character used to delimit constructed entries in the Classpath for the given operating system and Java Virtual Machine. If not defined, the operating system is queried for its default Classpath separator.

JAVA_CLASSPATH_DEFAULT A list of path names to .jar files to be added to the Java Classpath by default. The comma and/or space character delimits list entries.

JAVA_EXTRA_ARGUMENTS A list of additional arguments to be passed to the Java executable.

These macros control the power management capabilities of the *condor_startd* to optionally put the machine in to a low power state and wake it up later. See section 3.16 on page 471 on Power Management for more details.

HIBERNATE_CHECK_INTERVAL An integer number of seconds that determines how often the *condor_startd* checks to see if the machine is ready to enter a low power state. The default value is 0, which disables the check. If not 0, the HIBERNATE expression is evaluated within the context of each slot at the given interval. If used, a value 300 (5 minutes) is recommended.

As a special case, the interval is ignored when the machine has just returned from a low power state (excluding shutdown (5)). In order to avoid machines from volleying between a running state and a low power state, an hour of uptime is enforced after a machine has been woken. After the hour has passed, regular checks resume.

HIBERNATE A string expression that represents lower power state. When this state name evaluates to a valid non-"NONE" state (see below), causes Condor to put the machine into the specified low power state. The following names are supported (and are not case sensitive):

"NONE", "0": No-op: do not enter a low power state

"S1", "1", "STANDBY", "SLEEP": On Windows, this is Sleep (standby)

"S2", "2": On Windows, this is Sleep (standby)

"S3", "3", "RAM", "MEM", "SUSPEND": On Windows, this is Sleep (standby)

"S4", "4", "DISK", "HIBERNATE": Hibernate

"S5", "5", "SHUTDOWN", "OFF": Shutdown (soft-off)

The HIBERNATE expression is written in terms of the S-states as defined in the Advanced Configuration and Power Interface (ACPI) specification. The S-states take the form Sn, where n is an integer in the range 0 to 5, inclusive. The number that results from evaluating the expression determines which S-state to enter. The n from Sn notation was adopted because at this junction in time it appears to be the standard naming scheme for power states on several popular Operating Systems, including various flavors of Windows and Linux distributions. The other strings ("RAM", "DISK", etc.) are provided for ease of configuration.

Since this expression is evaluated in the context of each slot on the machine, any one slot has veto power over the other slots. If the evaluation of HIBERNATE in one slot evaluates to "NONE" or "0", then the machine will not be placed into a low power state. On the other hand, if all slots evaluate to a non-zero value, but differ in value, then the largest value is used as the representative power state.

Strings that do not match any in the table above are treated as "NONE".

UNHIBERNATE A boolean expression that specifies when an offline machine should be woken up. The default value is MachineLastMatchTime =!= UNDEFINED. This expression does not do anything, unless there is an instance of *condor_rooster* running, or another program that evaluates the Unhibernate expression of offline machine ClassAds. In addition, the collecting of offline machine ClassAds must be enabled for this expression to work. The variable OFFLINE_LOG , as detailed on page 206 explains this. The special attribute MachineLastMatchTime is updated in the ClassAds of offline machines when a job would have been matched to the machine if it had been online. For multi-slot machines, the offline ClassAd for slot1 will also contain the attributes slot<X>_MachineLastMatchTime, where X is replaced by the slot id of the other slots that would have been matched while offline. This allows the slot1 UNHIBERNATE expression

to refer to all of the slots on the machine, in case that is necessary. By default, *condor_rooster* will wake up a machine if any slot on the machine has its UNHIBERNATE expression evaluate to True.

HIBERNATION_PLUGIN A string which specifies the path and executable name of the hibernation plug-in that the *condor_startd* should use in the detection of low power states and switching to the low power states. The default value is $(LIBEXEC)/power_state. A default executable in that location which meets these specifications is shipped with Condor.

The *condor_startd* initially invokes this plug-in with the argument *ad*, and expects the plug-in to output a ClassAd to its standard output stream. The *condor_startd* will use this ClassAd to determine what low power setting to use on further invocations of the plug-in. To that end, the ClassAd must contain the attribute HibernationSupportedStates, a comma separated list of low power modes that are available. The recognized mode strings are the same as those in the table for the configuration variable HIBERNATE. The optional attribute HibernationMethod specifies a string which describes the mechanism used by the plug-in. The default Linux plug-in shipped with Condor will produce one of the strings NONE, /sys, /proc, or pm-utils. The optional attribute HibernationRawMask is an integer which represents the bit mask of the modes detected.

Subsequent *condor_startd* invocations of the plug-in have command line arguments **set** <*power-mode*>, where <*power-mode*> is one of the supported states as given in the attribute HibernationSupportedStates.

HIBERNATION_PLUGIN_ARGS Command line arguments appended to the command that invokes the plug-in. Used only when the *condor_startd* is *not* initially invoking the plug-in to determine the low power states supported.

HIBERNATION_OVERRIDE_WOL A boolean value that defaults to False. When True, it causes the *condor_startd* daemon's detection of the whether or not the network interface handles WOL packets to be ignored. When False, hibernation is disabled if the network interface does not use WOL packets to wake from hibernation. Therefore, when True hibernation can be enabled despite the fact that WOL packets are not used to wake machines.

LINUX_HIBERNATION_METHOD A string that can be used to override the default search used by Condor on Linux platforms to detect the hibernation method to use. This is used by the default hibernation plug-in executable that is shipped with Condor. The default behavior orders its search with:

1. Detect and use the *pm-utils* command line tools. The corresponding string is defined with "pm-utils".
2. Detect and use the directory in the virtual file system /sys/power. The corresponding string is defined with "/sys".
3. Detect and use the directory in the virtual file system /proc/ACPI. The corresponding string is defined with "/proc".

To override this ordered search behavior, and force the use of one particular method, set LINUX_HIBERNATION_METHOD to one of the defined strings.

OFFLINE_LOG The full path and file name of a file that stores machine ClassAds for every hibernating machine. This forms a persistent storage of these ClassAds, in case the *condor_collector* daemon crashes.

To avoid *condor_preen* removing this log, place it in a directory other than the directory defined by `$(SPOOL)`. Alternatively, if this log file is to go in the directory defined by `$(SPOOL)`, add the file to the list given by `VALID_SPOOL_FILES`.

OFFLINE_EXPIRE_ADS_AFTER An integer number of seconds specifying the lifetime of the persistent machine ClassAd representing a hibernating machine. Defaults to the largest 32-bit integer.

The following macros control the optional computation of resource availability statistics in the *condor_startd*.

STARTD_COMPUTE_AVAIL_STATS A boolean value that determines if the *condor_startd* computes resource availability statistics. The default is `False`.

If `STARTD_COMPUTE_AVAIL_STATS` is `True`, the *condor_startd* will define the following ClassAd attributes for resources:

AvailTime The proportion of the time (between 0.0 and 1.0) that this resource has been in a state other than Owner.

LastAvailInterval The duration in seconds of the last period between Owner states.

The following attributes will also be included if the resource is not in the Owner state:

AvailSince The time at which the resource last left the Owner state. Measured in the number of seconds since the epoch (00:00:00 UTC, Jan 1, 1970).

AvailTimeEstimate Based on past history, an estimate of how long the current period between Owner states will last.

STARTD_AVAIL_CONFIDENCE A floating point number representing the confidence level of the *condor_startd* daemon's `AvailTime` estimate. By default, the estimate is based on the 80th percentile of past values, so the value is initially set to 0.8.

STARTD_MAX_AVAIL_PERIOD_SAMPLES An integer that limits the number of samples of past available intervals stored by the *condor_startd* to limit memory and disk consumption. Each sample requires 4 bytes of memory and approximately 10 bytes of disk space.

3.3.11 condor_schedd Configuration File Entries

These macros control the *condor_schedd*.

SHADOW This macro determines the full path of the *condor_shadow* binary that the *condor_schedd* spawns. It is normally defined in terms of `$(SBIN)`.

START_LOCAL_UNIVERSE A boolean value that defaults to `TotalLocalJobsRunning < 200`. The *condor_schedd* uses this macro to determine whether to start a **local** universe job. At intervals determined by `SCHEDD_INTERVAL`, the *condor_schedd* daemon evaluates this macro for each idle **local** universe job that it has. For each job, if the `START_LOCAL_UNIVERSE` macro is `True`, then the job's `Requirements` expression is evaluated. If both conditions are met, then the job is allowed to begin execution.

The following example only allows 10 **local** universe jobs to execute concurrently. The attribute `TotalLocalJobsRunning` is supplied by *condor_schedd*'s ClassAd:

```
START_LOCAL_UNIVERSE = TotalLocalJobsRunning < 10
```

STARTER_LOCAL The complete path and executable name of the *condor_starter* to run for **local** universe jobs. This variable's value is defined in the initial configuration provided with Condor as

```
STARTER_LOCAL = $(SBIN)/condor_starter
```

This variable would only be modified or hand added into the configuration for a pool to be upgraded from one running a version of Condor that existed before the **local** universe to one that includes the **local** universe, but without utilizing the newer, provided configuration files.

LOCAL_UNIV_EXECUTE A string value specifying the execute location for local universe jobs. Each running local universe job will receive a uniquely named subdirectory within this directory. If not specified, it defaults to `$(SPOOL)/local_univ_execute`.

START_SCHEDULER_UNIVERSE A boolean value that defaults to `TotalSchedulerJobsRunning < 200`. The *condor_schedd* uses this macro to determine whether to start a **scheduler** universe job. At intervals determined by `SCHEDD_INTERVAL`, the *condor_schedd* daemon evaluates this macro for each idle **scheduler** universe job that it has. For each job, if the `START_SCHEDULER_UNIVERSE` macro is `True`, then the job's `Requirements` expression is evaluated. If both conditions are met, then the job is allowed to begin execution.

The following example only allows 10 **scheduler** universe jobs to execute concurrently. The attribute `TotalSchedulerJobsRunning` is supplied by *condor_schedd*'s ClassAd:

```
START_SCHEDULER_UNIVERSE = TotalSchedulerJobsRunning < 10
```

MAX_JOBS_RUNNING An integer representing a limit on the number of processes spawned by a given *condor_schedd* daemon, for all job universes except the grid universe. The number of processes limit includes *condor_shadow* processes, scheduler universe processes, including *condor_dagman*, and local universe *condor_starter* processes. Limiting the number of running scheduler and local universe jobs below the upper limit set by `MAX_JOBS_RUNNING` is best done using `START_LOCAL_UNIVERSE` and `START_SCHEDULER_UNIVERSE`. The

actual number of allowed *condor_shadow* daemons may be reduced, if the amount of memory defined by RESERVED_SWAP limits the number of *condor_shadow* daemons. A value for MAX_JOBS_RUNNING that is less than or equal to 0 prevents any new job from starting. Changing this setting to be below the current number of jobs that are running will cause running jobs to be aborted until the number running is within the limit.

Like all integer configuration variables, MAX_JOBS_RUNNING may be a ClassAd expression that evaluates to an integer, and which refers to constants either directly or via macro substitution. The default value is an expression that depends on the total amount of memory and the operating system. The default expression requires 1MByte of RAM per running job on the submit machine. In some environments and configurations, this is overly generous and can be cut by as much as 50%. On Windows platforms, the number of running jobs is still capped at 200. A 64-bit version of Windows is recommended in order to raise the value above the default. Under Unix, the maximum default is now 10,000. To scale higher, we recommend that the system ephemeral port range is extended such that there are at least 2.1 ports per running job.

Here are example configurations:

```
## Example 1:
MAX_JOBS_RUNNING = 10000

## Example 2:
## This is more complicated, but it produces the same limit as the default.
## First define some expressions to use in our calculation.
## Assume we can use up to 80% of memory and estimate shadow private data
## size of 800k.
MAX_SHADOWS_MEM = ceiling($(DETECTED_MEMORY)*0.8*1024/800)
## Assume we can use ~21,000 ephemeral ports (avg ~2.1 per shadow).
## Under Linux, the range is set in /proc/sys/net/ipv4/ip_local_port_range.
MAX_SHADOWS_PORTS = 10000
## Under windows, things are much less scalable, currently.
## Note that this can probably be safely increased a bit under 64-bit windows.
MAX_SHADOWS_OPSYS = ifThenElse(regexp("WIN.*","$(OPSYS)"),200,100000)
## Now build up the expression for MAX_JOBS_RUNNING.  This is complicated
## due to lack of a min() function.
MAX_JOBS_RUNNING = $(MAX_SHADOWS_MEM)
MAX_JOBS_RUNNING = \
  ifThenElse( $(MAX_SHADOWS_PORTS) < $(MAX_JOBS_RUNNING), \
              $(MAX_SHADOWS_PORTS), \
              $(MAX_JOBS_RUNNING) )
MAX_JOBS_RUNNING = \
  ifThenElse( $(MAX_SHADOWS_OPSYS) < $(MAX_JOBS_RUNNING), \
              $(MAX_SHADOWS_OPSYS), \
              $(MAX_JOBS_RUNNING) )
```

MAX_JOBS_SUBMITTED This integer value limits the number of jobs permitted in a *condor_schedd* daemon's queue. Submission of a new cluster of jobs fails, if the total number of jobs would exceed this limit. The default value for this variable is the largest positive integer value.

MAX_SHADOW_EXCEPTIONS This macro controls the maximum number of times that *condor_shadow* processes can have a fatal error (exception) before the *condor_schedd* will relinquish the match associated with the dying shadow. Defaults to 5.

MAX_PENDING_STARTD_CONTACTS An integer value that limits the number of simultaneous connection attempts by the *condor_schedd* when it is requesting claims from one or more *condor_startd* daemons. The intention is to protect the *condor_schedd* from being overloaded by authentication operations. The default value is 0. The special value 0 indicates no limit.

MAX_CONCURRENT_DOWNLOADS This specifies the maximum number of simultaneous transfers of output files from execute machines to the submit machine. The limit applies to all jobs submitted from the same *condor_schedd*. The default is 10. A setting of 0 means unlimited transfers. This limit currently does not apply to grid universe jobs or standard universe jobs, and it also does not apply to streaming output files. When the limit is reached, additional transfers will queue up and wait before proceeding.

MAX_CONCURRENT_UPLOADS This specifies the maximum number of simultaneous transfers of input files from the submit machine to execute machines. The limit applies to all jobs submitted from the same *condor_schedd*. The default is 10. A setting of 0 means unlimited transfers. This limit currently does not apply to grid universe jobs or standard universe jobs. When the limit is reached, additional transfers will queue up and wait before proceeding.

SCHEDD_QUERY_WORKERS This specifies the maximum number of concurrent sub-processes that the *condor_schedd* will spawn to handle queries. The setting is ignored in Windows. In Unix, the default is 3. If the limit is reached, the next query will be handled in the *condor_schedd*'s main process.

SCHEDD_INTERVAL This macro determines the maximum interval for both how often the *condor_schedd* sends a ClassAd update to the *condor_collector* and how often the *condor_schedd* daemon evaluates jobs. It is defined in terms of seconds and defaults to 300 (every 5 minutes).

SCHEDD_INTERVAL_TIMESLICE The bookkeeping done by the *condor_schedd* takes more time when there are large numbers of jobs in the job queue. However, when it is not too expensive to do this bookkeeping, it is best to keep the collector up to date with the latest state of the job queue. Therefore, this macro is used to adjust the bookkeeping interval so that it is done more frequently when the cost of doing so is relatively small, and less frequently when the cost is high. The default is 0.05, which means the schedd will adapt its bookkeeping interval to consume no more than 5% of the total time available to the schedd. The lower bound is configured by `SCHEDD_MIN_INTERVAL` (default 5 seconds), and the upper bound is configured by `SCHEDD_INTERVAL` (default 300 seconds).

JOB_START_COUNT This macro works together with the `JOB_START_DELAY` macro to throttle job starts. The default and minimum values for this integer configuration variable are both 1.

JOB_START_DELAY This integer-valued macro works together with the `JOB_START_COUNT` macro to throttle job starts. The *condor_schedd* daemon starts `$(JOB_START_COUNT)` jobs at a time, then delays for `$(JOB_START_DELAY)` seconds before starting the next set of jobs. This delay prevents a sudden, large load on resources required by the jobs during their start up phase. The resulting job start rate averages as fast as (`$(JOB_START_COUNT)`/`$(JOB_START_DELAY)`) jobs/second. This setting is defined

in terms of seconds and defaults to 0, which means jobs will be started as fast as possible. If you wish to throttle the rate of specific types of jobs, you can use the job attribute `NextJobStartDelay`.

MAX_NEXT_JOB_START_DELAY An integer number of seconds representing the maximum allowed value of the job ClassAd attribute `NextJobStartDelay`. It defaults to 600, which is 10 minutes.

JOB_STOP_COUNT An integer value representing the number of jobs operated on at one time by the *condor_schedd* daemon, when throttling the rate at which jobs are stopped via *condor_rm*, *condor_hold*, or *condor_vacate_job*. The default and minimum values are both 1. This variable is ignored for grid and scheduler universe jobs.

JOB_STOP_DELAY An integer value representing the number of seconds delay utilized by the *condor_schedd* daemon, when throttling the rate at which jobs are stopped via *condor_rm*, *condor_hold*, or *condor_vacate_job*. The *condor_schedd* daemon stops `$(JOB_STOP_COUNT)` jobs at a time, then delays for `$(JOB_STOP_DELAY)` seconds before stopping the next set of jobs. This delay prevents a sudden, large load on resources required by the jobs when they are terminating. The resulting job stop rate averages as fast as `JOB_STOP_COUNT/JOB_STOP_DELAY` jobs per second. This configuration variable is also used during the graceful shutdown of the *condor_schedd* daemon. During graceful shutdown, this macro determines the wait time in between requesting each *condor_shadow* daemon to gracefully shut down. The default value is 0, which means jobs will be stopped as fast as possible. This variable is ignored for grid and scheduler universe jobs.

JOB_IS_FINISHED_INTERVAL The *condor_schedd* maintains a list of jobs that are ready to permanently leave the job queue, e.g. they have completed or been removed. This integer-valued macro specifies a delay in seconds to place between the taking jobs permanently out of the queue. The default value is 0, which tells the *condor_schedd* to not impose any delay.

ALIVE_INTERVAL An initial value for an integer number of seconds defining how often the *condor_schedd* sends a UDP keep alive message to any *condor_startd* it has claimed. When the *condor_schedd* claims a *condor_startd*, the *condor_schedd* tells the *condor_startd* how often it is going to send these messages. The utilized interval for sending keep alive messages is the smallest of the two values `ALIVE_INTERVAL` and the expression `JobLeaseDuration/3`, formed with the job ClassAd attribute `JobLeaseDuration`. The value of the interval is further constrained by the floor value of 10 seconds. If the *condor_startd* does not receive any of these keep alive messages during a certain period of time (defined via `MAX_CLAIM_ALIVES_MISSED` , described on page 199) the *condor_startd* releases the claim, and the *condor_schedd* no longer pays for the resource (in terms of user priority in the system). The macro is defined in terms of seconds and defaults to 300, which is 5 minutes.

STARTD_SENDS_ALIVES A boolean value that defaults to `True`, causing keep alive messages to be sent from the *condor_startd* to the *condor_schedd* by TCP during a claim. When `False`, the *condor_schedd* daemon sends keep alive signals to the the *condor_startd*, reversing the direction. If both *condor_startd* and *condor_schedd* daemons are Condor version 7.5.4 or more recent, this variable is only used by the *condor_schedd* daemon. For earlier

Condor versions, the variable must be set to the same value, and it must be set for both daemons.

REQUEST_CLAIM_TIMEOUT This macro sets the time (in seconds) that the *condor_schedd* will wait for a claim to be granted by the *condor_startd*. The default is 30 minutes. This is only likely to matter if the *condor_startd* has an existing claim and it takes a long time for the existing claim to be preempted due to `MaxJobRetirementTime`. Once a request times out, the *condor_schedd* will simply begin the process of finding a machine for the job all over again.

Normally, it is not a good idea to set this to be very small (e.g. a few minutes). Doing so can lead to failure to preempt, because the preempting job will spend a significant fraction of its time waiting to be re-matched. During that time, it would miss out on any opportunity to run if the job it is trying to preempt gets out of the way.

SHADOW_SIZE_ESTIMATE The estimated private virtual memory size of each *condor_shadow* process in Kbytes. This value is only used if `RESERVED_SWAP` is non-zero. The default value is 800.

SHADOW_RENICE_INCREMENT When the *condor_schedd* spawns a new *condor_shadow*, it can do so with a *nice-level*. A nice-level is a Unix mechanism that allows users to assign their own processes a lower priority so that the processes run with less priority than other tasks on the machine. The value can be any integer between 0 and 19, with a value of 19 being the lowest priority. It defaults to 0.

SCHED_UNIV_RENICE_INCREMENT Analogous to `JOB_RENICE_INCREMENT` and `SHADOW_RENICE_INCREMENT`, scheduler universe jobs can be given a nice-level. The value can be any integer between 0 and 19, with a value of 19 being the lowest priority. It defaults to 0.

QUEUE_CLEAN_INTERVAL The *condor_schedd* maintains the job queue on a given machine. It does so in a persistent way such that if the *condor_schedd* crashes, it can recover a valid state of the job queue. The mechanism it uses is a transaction-based log file (the `job_queue.log` file, not the `SchedLog` file). This file contains an initial state of the job queue, and a series of transactions that were performed on the queue (such as new jobs submitted, jobs completing, and checkpointing). Periodically, the *condor_schedd* will go through this log, truncate all the transactions and create a new file with containing only the new initial state of the log. This is a somewhat expensive operation, but it speeds up when the *condor_schedd* restarts since there are fewer transactions it has to play to figure out what state the job queue is really in. This macro determines how often the *condor_schedd* should rework this queue to cleaning it up. It is defined in terms of seconds and defaults to 86400 (once a day).

WALL_CLOCK_CKPT_INTERVAL The job queue contains a counter for each job's "wall clock" run time, i.e., how long each job has executed so far. This counter is displayed by *condor_q*. The counter is updated when the job is evicted or when the job completes. When the *condor_schedd* crashes, the run time for jobs that are currently running will not be added to the counter (and so, the run time counter may become smaller than the CPU time counter). The *condor_schedd* saves run time "checkpoints" periodically for running jobs so if the *condor_schedd* crashes, only run time since the last checkpoint is lost. This macro controls how

often the *condor_schedd* saves run time checkpoints. It is defined in terms of seconds and defaults to 3600 (one hour). A value of 0 will disable wall clock checkpoints.

QUEUE_ALL_USERS_TRUSTED Defaults to False. If set to True, then unauthenticated users are allowed to write to the queue, and also we always trust whatever the `Owner` value is set to be by the client in the job ad. This was added so users can continue to use the SOAP web-services interface over HTTP (w/o authenticating) to submit jobs in a secure, controlled environment – for instance, in a portal setting.

QUEUE_SUPER_USERS A comma and/or space separated list of user names on a given machine that are given *super-user access* to the job queue, meaning that they can modify or delete the job ClassAds of other users. When not on this list, users can only modify or delete their own ClassAds from the job queue. Whatever user name corresponds with the UID that Condor is running as – usually user `condor` – will automatically be included in this list, because that is needed for Condor's proper functioning. See section 3.6.13 on UIDs in Condor for more details on this. By default, the Unix user `root` and the Windows user `administrator` are given the ability to remove other user's jobs, in addition to user `condor`.

SYSTEM_JOB_MACHINE_ATTRS This macro specifies a space and/or comma separated list of machine attributes that should be recorded in the job ClassAd. The default attributes are `Cpus` and `SlotWeight`. When there are multiple run attempts, history of machine attributes from previous run attempts may be kept. The number of run attempts to store is specified by the configuration variable `SYSTEM_JOB_MACHINE_ATTRS_HISTORY_LENGTH` . A machine attribute named `X` will be inserted into the job ClassAd as an attribute named `MachineAttrX0`. The previous value of this attribute will be named `MachineAttrX1`, the previous to that will be named `MachineAttrX2`, and so on, up to the specified history length. A history of length 1 means that only `MachineAttrX0` will be recorded. Additional attributes to record may be specified on a per-job basis by using the **job_machine_attrs** submit file command. The value recorded in the job ClassAd is the evaluation of the machine attribute in the context of the job ClassAd when the *condor_schedd* daemon initiates the start up of the job. If the evaluation results in an `Undefined` or `Error` result, the value recorded in the job ClassAd will be `Undefined` or `Error` respectively.

SYSTEM_JOB_MACHINE_ATTRS_HISTORY_LENGTH The integer number of run attempts to store in the job ClassAd when recording the values of machine attributes listed in `SYSTEM_JOB_MACHINE_ATTRS` . The default is 1. The history length may also be extended on a per-job basis by using the submit file command **job_machine_attrs_history_length**. The larger of the system and per-job history lengths will be used. A history length of 0 disables recording of machine attributes.

SCHEDD_LOCK This macro specifies what lock file should be used for access to the `SchedLog` file. It must be a separate file from the `SchedLog`, since the `SchedLog` may be rotated and synchronization across log file rotations is desired. This macro is defined relative to the `$(LOCK)` macro.

SCHEDD_NAME Used to give an alternative value to the `Name` attribute in the *condor_schedd*'s ClassAd.

See the description of MASTER_NAME in section 3.3.9 on page 194 for defaults and composition of valid Condor daemon names. Also, note that if the MASTER_NAME setting is defined for the *condor_master* that spawned a given *condor_schedd*, that name will take precedence over whatever is defined in SCHEDD_NAME.

SCHEDD_ATTRS This macro is described in section 3.3.5 as <SUBSYS>_ATTRS.

SCHEDD_DEBUG This macro (and other settings related to debug logging in the *condor_schedd*) is described in section 3.3.4 as <SUBSYS>_DEBUG.

SCHEDD_ADDRESS_FILE This macro is described in section 3.3.5 as <SUBSYS>_ADDRESS_FILE.

SCHEDD_EXECUTE A directory to use as a temporary sandbox for local universe jobs. Defaults to $(SPOOL)/execute.

FLOCK_NEGOTIATOR_HOSTS Defines a comma and/or space separated list of *condor_negotiator* host names for pools in which the *condor_schedd* should attempt to run jobs. If not set, the *condor_schedd* will query the *condor_collector* daemons for the addresses of the *condor_negotiator* daemons. If set, then the *condor_negotiator* daemons must be specified in order, corresponding to the list set by FLOCK_COLLECTOR_HOSTS. In the typical case, where each pool has the *condor_collector* and *condor_negotiator* running on the same machine, $(FLOCK_NEGOTIATOR_HOSTS) should have the same definition as $(FLOCK_COLLECTOR_HOSTS). This configuration value is also typically used as a macro for adding the *condor_negotiator* to the relevant authorization lists.

FLOCK_COLLECTOR_HOSTS This macro defines a list of collector host names (not including the local $(COLLECTOR_HOST) machine) for pools in which the *condor_schedd* should attempt to run jobs. Hosts in the list should be in order of preference. The *condor_schedd* will only send a request to a central manager in the list if the local pool and pools earlier in the list are not satisfying all the job requests. $(HOSTALLOW_NEGOTIATOR_SCHEDD) (see section 3.3.5) must also be configured to allow negotiators from all of the pools to contact the *condor_schedd* at the NEGOTIATOR authorization level. Similarly, the central managers of the remote pools must be configured to allow this *condor_schedd* to join the pool (this requires ADVERTISE_SCHEDD authorization level, which defaults to WRITE).

NEGOTIATE_ALL_JOBS_IN_CLUSTER If this macro is set to False (the default), when the *condor_schedd* fails to start an idle job, it will not try to start any other idle jobs in the same cluster during that negotiation cycle. This makes negotiation much more efficient for large job clusters. However, in some cases other jobs in the cluster can be started even though an earlier job can't. For example, the jobs' requirements may differ, because of different disk space, memory, or operating system requirements. Or, machines may be willing to run only some jobs in the cluster, because their requirements reference the jobs' virtual memory size or other attribute. Setting this macro to True will force the *condor_schedd* to try to start all idle jobs in each negotiation cycle. This will make negotiation cycles last longer, but it will ensure that all jobs that can be started will be started.

PERIODIC_EXPR_INTERVAL This macro determines the minimum period, in seconds, between evaluation of periodic job control expressions, such as periodic_hold, periodic_release, and

periodic_remove, given by the user in a Condor submit file. By default, this value is 60 seconds. A value of 0 prevents the *condor_schedd* from performing the periodic evaluations.

MAX_PERIODIC_EXPR_INTERVAL This macro determines the maximum period, in seconds, between evaluation of periodic job control expressions, such as periodic_hold, periodic_release, and periodic_remove, given by the user in a Condor submit file. By default, this value is 1200 seconds. If Condor is behind on processing events, the actual period between evaluations may be higher than specified.

PERIODIC_EXPR_TIMESLICE This macro is used to adapt the frequency with which the *condor_schedd* evaluates periodic job control expressions. When the job queue is very large, the cost of evaluating all of the ClassAds is high, so in order for the *condor_schedd* to continue to perform well, it makes sense to evaluate these expressions less frequently. The default time slice is 0.01, so the *condor_schedd* will set the interval between evaluations so that it spends only 1% of its time in this activity. The lower bound for the interval is configured by `PERIODIC_EXPR_INTERVAL` (default 60 seconds) and the upper bound is configured with `MAX_PERIODIC_EXPR_INTERVAL` (default 1200 seconds).

SYSTEM_PERIODIC_HOLD This expression behaves identically to the job expression `periodic_hold`, but it is evaluated by the *condor_schedd* daemon individually for each job in the queue. It defaults to `False`. When `True`, it causes the job to stop running and go on hold. Here is an example that puts jobs on hold if they have been restarted too many times, have an unreasonably large virtual memory `ImageSize`, or have unreasonably large disk usage for an invented environment.

```
SYSTEM_PERIODIC_HOLD = \
  (JobStatus == 1 || JobStatus == 2) && \
  (JobRunCount > 10 || ImageSize > 3000000 || DiskUsage > 10000000)
```

SYSTEM_PERIODIC_RELEASE This expression behaves identically to the job expression `periodic_release`, but it is evaluated by the *condor_schedd* daemon individually for each job in the queue. It defaults to `False`. When `True`, it causes a held job to return to the idle state. Here is an example that releases jobs from hold if they have tried to run less than 20 times, have most recently been on hold for over 20 minutes, and have gone on hold due to "Connection timed out" when trying to execute the job, because the file system containing the job's executable is temporarily unavailable.

```
SYSTEM_PERIODIC_RELEASE = \
  (JobRunCount < 20 && CurrentTime - EnteredCurrentStatus > 1200 ) && ( \
    (HoldReasonCode == 6 && HoldReasonSubCode == 110) \
  )
```

SYSTEM_PERIODIC_REMOVE This expression behaves identically to the job expression `periodic_remove`, but it is evaluated by the *condor_schedd* daemon individually for each job in the queue. It defaults to `False`. When `True`, it causes the job to be removed from the queue. Here is an example that removes jobs which have been on hold for 30 days:

```
SYSTEM_PERIODIC_REMOVE = \
  (JobStatus == 5 && CurrentTime - EnteredCurrentStatus > 3600*24*30)
```

SCHEDD_ASSUME_NEGOTIATOR_GONE This macro determines the period, in seconds, that the *condor_schedd* will wait for the *condor_negotiator* to initiate a negotiation cycle before the schedd will simply try to claim any local *condor_startd*. This allows for a machine that is acting as both a submit and execute node to run jobs locally if it cannot communicate with the central manager. The default value, if not specified, is 4 x `$(NEGOTIATOR_INTERVAL)`. If `$(NEGOTIATOR_INTERVAL)` is not defined, then `SCHEDD_ASSUME_NEGOTIATOR_GONE` will default to 1200 (20 minutes).

SCHEDD_ROUND_ATTR_<xxxx> This is used to round off attributes in the job ClassAd so that similar jobs may be grouped together for negotiation purposes. There are two cases. One is that a percentage such as 25% is specified. In this case, the value of the attribute named <xxxx>\ in the job ClassAd will be rounded up to the next multiple of the specified percentage of the values order of magnitude. For example, a setting of 25% will cause a value near 100 to be rounded up to the next multiple of 25 and a value near 1000 will be rounded up to the next multiple of 250. The other case is that an integer, such as 4, is specified instead of a percentage. In this case, the job attribute is rounded up to the specified number of decimal places. Replace <xxxx> with the name of the attribute to round, and set this macro equal to the number of decimal places to round up. For example, to round the value of job ClassAd attribute `foo` up to the nearest 100, set

```
SCHEDD_ROUND_ATTR_foo = 2
```

When the schedd rounds up an attribute value, it will save the raw (un-rounded) actual value in an attribute with the same name appended with "_RAW". So in the above example, the raw value will be stored in attribute `foo_RAW` in the job ClassAd. The following are set by default:

```
SCHEDD_ROUND_ATTR_ImageSize = 25%
SCHEDD_ROUND_ATTR_ExecutableSize = 25%
SCHEDD_ROUND_ATTR_DiskUsage = 25%
SCHEDD_ROUND_ATTR_NumCkpts = 4
```

Thus, an ImageSize near 100MB will be rounded up to the next multiple of 25MB. If your batch slots have less memory or disk than the rounded values, it may be necessary to reduce the amount of rounding, because the job requirements will not be met.

SCHEDD_BACKUP_SPOOL This macro is used to enable the *condor_schedd* to make a backup of the job queue as it starts. If set to "True", the *condor_schedd* will create host specific a backup of the current spool file to the spool directory. This backup file will be overwritten each time the *condor_schedd* starts. `SCHEDD_BACKUP_SPOOL` defaults to "False".

SCHEDD_PREEMPTION_REQUIREMENTS This boolean expression is utilized only for machines allocated by a dedicated scheduler. When `True`, a machine becomes a candidate for job preemption. This configuration variable has no default; when not defined, preemption will never be considered.

SCHEDD_PREEMPTION_RANK This floating point value is utilized only for machines allocated by a dedicated scheduler. It is evaluated in context of a job ClassAd, and it represents a

machine's preference for running a job. This configuration variable has no default; when not defined, preemption will never be considered.

ParallelSchedulingGroup For parallel jobs which must be assigned within a group of machines (and not cross group boundaries), this configuration variable identifies members of a group. Each machine within a group sets this configuration variable with a string that identifies the group.

PER_JOB_HISTORY_DIR If set to a directory writable by the Condor user, when a job leaves the *condor_schedd*'s queue, a copy of its ClassAd will be written in that directory. The files are named "history." with the job's cluster and process number appended. For example, job 35.2 will result in a file named "history.35.2". Condor does not rotate or delete the files, so without an external entity to clean the directory it can grow very large. This option defaults to being unset. When not set, no such files are written.

DEDICATED_SCHEDULER_USE_FIFO When this parameter is set to true (the default), parallel universe jobs will be scheduled in a first-in, first-out manner. When set to false, parallel jobs are scheduled using a best-fit algorithm. Using the best-fit algorithm is not recommended, as it can cause starvation.

SCHEDD_SEND_VACATE_VIA_TCP A boolean value that defaults to `False`. When `True`, the *condor_schedd* daemon sends vacate signals via TCP, instead of the default UDP.

SCHEDD_CLUSTER_INITIAL_VALUE An integer that specifies the initial cluster number value to use within a job id when a job is first submitted. The default value is 1.

SCHEDD_CLUSTER_INCREMENT_VALUE A positive integer that defaults to 1, representing a stride used for assignment of cluster numbers within a job id. When a job is submitted, the job will be assigned a job id. The cluster number of the job id will be equal to the previous cluster number used plus the value of this setting.

SCHEDD_CLUSTER_MAXIMUM_VALUE An integer that specifies an upper bound on assigned job cluster id values. For value M, the maximum job cluster id assigned to any job will be $M - 1$. When the maximum id is reached, cluster ids will continue assignment using `SCHEDD_CLUSTER_INITIAL_VALUE` . The default value of this variable is zero, which represents the behavior of having no maximum cluster id value.

Note that Condor does not check for nor take responsibility for duplicate cluster ids for queued jobs. If `SCHEDD_CLUSTER_MAXIMUM_VALUE` is set to a non-zero value, the system administrator is responsible for ensuring that older jobs do not stay in the queue long enough for cluster ids of new jobs to wrap around and reuse the same id. With a low enough value, it is possible for jobs to be erroneously assigned duplicate cluster ids, which will result in a corrupt job queue.

GRIDMANAGER_SELECTION_EXPR By default, the *condor_schedd* daemon will start a new *condor_gridmanager* process for each discrete user that submits a grid universe job, that is, for each discrete value of job attribute `Owner` across all grid universe job ClassAds. For additional isolation and/or scalability of grid job management, additional *condor_gridmanager* processes can be spawned to share the load; to do so, set this variable to be a ClassAd expression. The result of the evaluation of this expression in the context of a grid universe

job ClassAd will be treated as a hash value. All jobs that hash to the same value via this expression will go to the same *condor_gridmanager*. For instance, to spawn a separate *condor_gridmanager* process to manage each unique remote site, the following expression works:

```
GRIDMANAGER_SELECTION_EXPR = GridResource
```

CKPT_SERVER_CLIENT_TIMEOUT An integer which specifies how long in seconds the *condor_schedd* is willing to wait for a response from a checkpoint server before declaring the checkpoint server down. The value of 0 makes the schedd block for the operating system configured time (which could be a very long time) before the `connect()` returns on its own with a connection timeout. The default value is 20.

CKPT_SERVER_CLIENT_TIMEOUT_RETRY An integer which specifies how long in seconds the *condor_schedd* will ignore a checkpoint server that is deemed to be down. After this time elapses, the *condor_schedd* will try again in talking to the checkpoint server. The default is 1200.

SCHEDD_JOB_QUEUE_LOG_FLUSH_DELAY An integer which specifies an upper bound in seconds on how long it takes for changes to the job ClassAd to be visible to the Condor Job Router and to Quill. The default is 5 seconds.

3.3.12 condor_shadow Configuration File Entries

These settings affect the *condor_shadow*.

SHADOW_LOCK This macro specifies the lock file to be used for access to the `ShadowLog` file. It must be a separate file from the `ShadowLog`, since the `ShadowLog` may be rotated and you want to synchronize access across log file rotations. This macro is defined relative to the `$(LOCK)` macro.

SHADOW_DEBUG This macro (and other settings related to debug logging in the shadow) is described in section 3.3.4 as `<SUBSYS>_DEBUG`.

SHADOW_QUEUE_UPDATE_INTERVAL The amount of time (in seconds) between ClassAd updates that the *condor_shadow* daemon sends to the *condor_schedd* daemon. Defaults to 900 (15 minutes).

SHADOW_LAZY_QUEUE_UPDATE This boolean macro specifies if the *condor_shadow* should immediately update the job queue for certain attributes (at this time, it only effects the `NumJobStarts` and `NumJobReconnects` counters) or if it should wait and only update the job queue on the next periodic update. There is a trade-off between performance and the semantics of these attributes, which is why the behavior is controlled by a configuration macro. If the *condor_shadow* do not use a lazy update, and immediately ensures the changes to the job attributes are written to the job queue on disk, the semantics for the attributes are very solid (there's only a tiny chance that the counters will be out of sync with reality), but this introduces a potentially large performance and scalability problem for a busy *condor_schedd*.

If the *condor_shadow* uses a lazy update, there's no additional cost to the *condor_schedd*, but it means that *condor_q* and Quill won't immediately see the changes to the job attributes, and if the *condor_shadow* happens to crash or be killed during that time, the attributes are never incremented. Given that the most obvious usage of these counter attributes is for the periodic user policy expressions (which are evaluated directly by the *condor_shadow* using its own copy of the job's classified ad, which is immediately updated in either case), and since the additional cost for aggressive updates to a busy *condor_schedd* could potentially cause major problems, the default is `True` to do lazy, periodic updates.

SHADOW_WORKLIFE The integer number of seconds after which the *condor_shadow* will exit when the current job finishes, instead of fetching a new job to manage. Having the *condor_shadow* continue managing jobs helps reduce overhead and can allow the *condor_schedd* to achieve higher job completion rates. The default is 3600, one hour. The value 0 causes *condor_shadow* to exit after running a single job.

COMPRESS_PERIODIC_CKPT A boolean value that when `True`, directs the *condor_shadow* to instruct applications to compress periodic checkpoints when possible. The default is `False`.

COMPRESS_VACATE_CKPT A boolean value that when `True`, directs the *condor_shadow* to instruct applications to compress vacate checkpoints when possible. The default is `False`.

PERIODIC_MEMORY_SYNC This boolean value specifies whether the *condor_shadow* should instruct applications to commit dirty memory pages to swap space during a periodic checkpoint. The default is `False`. This potentially reduces the number of dirty memory pages at vacate time, thereby reducing swapping activity on the remote machine.

SLOW_CKPT_SPEED This macro specifies the speed at which vacate checkpoints should be written, in kilobytes per second. If zero (the default), vacate checkpoints are written as fast as possible. Writing vacate checkpoints slowly can avoid overwhelming the remote machine with swapping activity.

SHADOW_JOB_CLEANUP_RETRY_DELAY This integer specifies the number of seconds to wait between tries to commit the final update to the job ClassAd in the *condor_schedd*'s job queue. The default is 30.

SHADOW_MAX_JOB_CLEANUP_RETRIES This integer specifies the number of times to try committing the final update to the job ClassAd in the *condor_schedd*'s job queue. The default is 5.

SHADOW_CHECKPROXY_INTERVAL The number of seconds between tests to see if the job proxy has been updated or should be refreshed. The default is 600 seconds (10 minutes). This variable's value should be small in comparison to the refresh interval required to keep delegated credentials from expiring (configured via `DELEGATE_JOB_GSI_CREDENTIALS_REFRESH` and `DELEGATE_JOB_GSI_CREDENTIALS_LIFETIME`). If this variable's value is too small, proxy updates could happen very frequently, potentially creating a lot of load on the submit machine.

3.3.13 condor_starter Configuration File Entries

These settings affect the *condor_starter*.

EXEC_TRANSFER_ATTEMPTS Sometimes due to a router misconfiguration, kernel bug, or other network problem, the transfer of the initial checkpoint from the submit machine to the execute machine will fail midway through. This parameter allows a retry of the transfer a certain number of times that must be equal to or greater than 1. If this parameter is not specified, or specified incorrectly, then it will default to three. If the transfer of the initial executable fails every attempt, then the job goes back into the idle state until the next renegotiation cycle.

NOTE: : This parameter does not exist in the NT starter.

JOB_RENICE_INCREMENT When the *condor_starter* spawns a Condor job, it can do so with a *nice-level*. A nice-level is a Unix mechanism that allows users to assign their own processes a lower priority, such that these processes do not interfere with interactive use of the machine. For machines with lots of real memory and swap space, such that the only scarce resource is CPU time, use this macro in conjunction with a policy that allows Condor to always start jobs on the machines. Condor jobs would always run, but interactive response on the machines would never suffer. A user most likely will not notice Condor is running jobs. See section 3.5 on Startd Policy Configuration for more details on setting up a policy for starting and stopping jobs on a given machine.

The ClassAd expression is evaluated in the context of the job ad to an integer value, which is set by the *condor_starter* daemon for each job just before the job runs. The range of allowable values are integers in the range of 0 to 19 (inclusive), with a value of 19 being the lowest priority. If the integer value is outside this range, then on a Unix machine, a value greater than 19 is auto-decreased to 19; a value less than 0 is treated as 0. For values outside this range, a Windows machine ignores the value and uses the default instead. The default value is 10, which maps to the idle priority class on a Windows machine.

STARTER_LOCAL_LOGGING This macro determines whether the starter should do local logging to its own log file, or send debug information back to the *condor_shadow* where it will end up in the ShadowLog. It defaults to `True`.

STARTER_DEBUG This setting (and other settings related to debug logging in the starter) is described above in section 3.3.4 as `$(<SUBSYS>_DEBUG)`.

STARTER_UPDATE_INTERVAL An integer value representing the number of seconds between ClassAd updates that the *condor_starter* daemon sends to the *condor_shadow* and *condor_startd* daemons. Defaults to 300 (5 minutes).

STARTER_UPDATE_INTERVAL_TIMESLICE A floating point value, specifying the highest fraction of time that the *condor_starter* daemon should spend collecting monitoring information about the job, such as disk usage. The default value is 0.1. If monitoring, such as checking disk usage takes a long time, the *condor_starter* will monitor less frequently than specified by `STARTER_UPDATE_INTERVAL`.

USER_JOB_WRAPPER The full path to an executable or script. This macro allows an administrator to specify a wrapper script to handle the execution of all user jobs. If specified, Condor never directly executes a job, but instead invokes the program specified by this macro. The command-line arguments passed to this program will include the full-path to the actual user job which should be executed, followed by all the command-line parameters to pass to the user job. This wrapper program must ultimately replace its image with the user job; in other words, it must `exec()` the user job, not `fork()` it. For instance, if the wrapper program is a C/Korn shell script, the last line of execution should be:

```
exec $*
```

This can potentially lose information about the arguments. Any argument with embedded white space will be split into multiple arguments. For example the argument "argument one" will become the two arguments "argument" and "one". For Bourne type shells (sh, bash, ksh), the following preserves the arguments:

```
exec "$@"
```

For the C type shells (csh, tcsh), the following preserves the arguments:

```
exec $*:q
```

For Windows machines, the wrapper will either be a batch script (with a file extension of `.bat` or `.cmd`) or an executable (with a file extension of `.exe` or `.com`).

USE_VISIBLE_DESKTOP This setting is only meaningful on Windows machines. If True, Condor will allow the job to create windows on the desktop of the execute machine and interact with the job. This is particularly useful for debugging why an application will not run under Condor. If False, Condor uses the default behavior of creating a new, non-visible desktop to run the job on. See section 6.2 for details on how Condor interacts with the desktop.

STARTER_JOB_ENVIRONMENT This macro sets the default environment inherited by jobs. The syntax is the same as the syntax for environment settings in the job submit file (see page 829). If the same environment variable is assigned by this macro and by the user in the submit file, the user's setting takes precedence.

JOB_INHERITS_STARTER_ENVIRONMENT A boolean value that defaults to `False`. When `True`, it causes jobs to inherit all environment variables from the *condor_starter*. This is useful for glidein jobs that need to access environment variables from the batch system running the glidein daemons. When both the user job and `STARTER_JOB_ENVIRONMENT` define an environment variable that is in the *condor_starter*'s environment, the user job's definition takes precedence. This variable does not apply to standard universe jobs.

STARTER_UPLOAD_TIMEOUT An integer value that specifies the network communication timeout to use when transferring files back to the submit machine. The default value is set by the *condor_shadow* daemon to 300. Increase this value if the disk on the submit machine cannot keep up with large bursts of activity, such as many jobs all completing at the same time.

ENFORCE_CPU_AFFINITY A boolean value that defaults to `False`. When `False`, the affinity of jobs and their descendants to a CPU is not enforced. When `True`, Condor jobs and their descendants maintain their affinity to a CPU. When `True`, more fine grained affinities may be specified with `SLOT<N>_CPU_AFFINITY`.

SLOT<N>_CPU_AFFINITY A comma separated list of cores to which a Condor job running on a specific slot given by the value of `<N>` show affinity. Note that slots are numbered beginning with the value 1, while CPU cores are numbered beginning with the value 0. This affinity list only takes effect if `ENFORCE_CPU_AFFINITY = True`.

ENABLE_URL_TRANSFERS A boolean value that when `True` causes the *condor_starter* for a job to invoke all plug-ins defined by `FILETRANSFER_PLUGINS` to determine their capabilities for handling protocols to be used in file transfer specified with a URL. When `False`, a URL transfer specified in a job's submit description file will cause an error issued by *condor_submit*. The default value is `True`.

FILETRANSFER_PLUGINS A comma separated list of full and absolute path and executable names for plug-ins that will accomplish the task of doing file transfer when a job requests the transfer of an input file by specifying a URL. See section 3.13.3 for a description of the functionality required of a plug-in.

ENABLE_CHIRP A boolean value that defaults to `True`. An administrator would set the value to `False` to disable Chirp remote file access from execute machines.

3.3.14 condor_submit Configuration File Entries

DEFAULT_UNIVERSE The universe under which a job is executed may be specified in the submit description file. If it is not specified in the submit description file, then this variable specifies the universe (when defined). If the universe is not specified in the submit description file, and if this variable is not defined, then the default universe for a job will be the vanilla universe.

If you want *condor_submit* to automatically append an expression to the `Requirements` expression or `Rank` expression of jobs at your site use the following macros:

APPEND_REQ_VANILLA Expression to be appended to vanilla job requirements.

APPEND_REQ_STANDARD Expression to be appended to standard job requirements.

APPEND_REQUIREMENTS Expression to be appended to any type of universe jobs. However, if `APPEND_REQ_VANILLA` or `APPEND_REQ_STANDARD` is defined, then ignore the `APPEND_REQUIREMENTS` for those universes.

APPEND_RANK Expression to be appended to job rank. `APPEND_RANK_STANDARD` or `APPEND_RANK_VANILLA` will override this setting if defined.

APPEND_RANK_STANDARD Expression to be appended to standard job rank.

APPEND_RANK_VANILLA Expression to append to vanilla job rank.

NOTE: The APPEND_RANK_STANDARD and APPEND_RANK_VANILLA macros were called APPEND_PREF_STANDARD and APPEND_PREF_VANILLA in previous versions of Condor.

In addition, you may provide default Rank expressions if your users do not specify their own with:

DEFAULT_RANK Default rank expression for any job that does not specify its own rank expression in the submit description file. There is no default value, such that when undefined, the value used will be 0.0.

DEFAULT_RANK_VANILLA Default rank for vanilla universe jobs. There is no default value, such that when undefined, the value used will be 0.0. When both DEFAULT_RANK and DEFAULT_RANK_VANILLA are defined, the value for DEFAULT_RANK_VANILLA is used for vanilla universe jobs.

DEFAULT_RANK_STANDARD Default rank for standard universe jobs. There is no default value, such that when undefined, the value used will be 0.0. When both DEFAULT_RANK and DEFAULT_RANK_STANDARD are defined, the value for DEFAULT_RANK_STANDARD is used for standard universe jobs.

DEFAULT_IO_BUFFER_SIZE Condor keeps a buffer of recently-used data for each file an application opens. This macro specifies the default maximum number of bytes to be buffered for each open file at the executing machine. The *condor_status* buffer_size command will override this default. If this macro is undefined, a default size of 512 KB will be used.

DEFAULT_IO_BUFFER_BLOCK_SIZE When buffering is enabled, Condor will attempt to consolidate small read and write operations into large blocks. This macro specifies the default block size Condor will use. The *condor_status* buffer_block_size command will override this default. If this macro is undefined, a default size of 32 KB will be used.

SUBMIT_SKIP_FILECHECKS If True, *condor_submit* behaves as if the **-d** command-line option is used. This tells *condor_submit* to disable file permission checks when submitting a job. This can significantly decrease the amount of time required to submit a large group of jobs. The default value is False.

WARN_ON_UNUSED_SUBMIT_FILE_MACROS A boolean variable that defaults to True. When True, *condor_submit* performs checks on the job's submit description file contents for commands that define a macro, but do not use the macro within the file. A warning is issued, but job submission continues. A definition of a new macro occurs when the lhs of a command is not a known submit command. This check may help spot spelling errors of known submit commands.

SUBMIT_SEND_RESCHEDULE A boolean expression that when False, prevents *condor_submit* from automatically sending a *condor_reschedule* command as it completes. The *condor_reschedule* command causes the *condor_schedd* daemon to start searching for machines

with which to match the submitted jobs. When True, this step always occurs. In the case that the machine where the job(s) are submitted is managing a huge number of jobs (thousands or tens of thousands), this step would hurt performance in such a way that it became an obstacle to scalability. The default value is True.

SUBMIT_EXPRS A comma-separated list of ClassAd attributes to be inserted into all the job ClassAds that *condor_submit* creates. This is equivalent to the "+" syntax in a submit description file. Attributes defined in the submit description file with "+" will override attributes defined in the configuration file with SUBMIT_EXPRS. Note that adding an attribute to a job's ClassAd will *not* function as a method for specifying default values of submit description file commands forgotten in a job's submit description file. The command in the submit description file results in actions by *condor_submit*, while the use of SUBMIT_EXPRS adds a job ClassAd attribute at a later point in time.

LOG_ON_NFS_IS_ERROR A boolean value that controls whether *condor_submit* prohibits job submit files with user log files on NFS. If LOG_ON_NFS_IS_ERROR is set to True, such submit files will be rejected. If LOG_ON_NFS_IS_ERROR is set to False, the job will be submitted. If not defined, LOG_ON_NFS_IS_ERROR defaults to False.

SUBMIT_MAX_PROCS_IN_CLUSTER An integer value that limits the maximum number of jobs that would be assigned within a single cluster. Job submissions that would exceed the defined value fail, issuing an error message, and with no jobs submitted. The default value is 0, which does not limit the number of jobs assigned a single cluster number.

3.3.15 condor_preen Configuration File Entries

These macros affect *condor_preen*.

PREEN_ADMIN This macro sets the e-mail address where *condor_preen* will send e-mail (if it is configured to send email at all; see the entry for PREEN). Defaults to $(CONDOR_ADMIN).

VALID_SPOOL_FILES This macro contains a (comma or space separated) list of files that *condor_preen* considers valid files to find in the $(SPOOL) directory. There is no default value. *condor_preen* will add to the list files and directories that are normally present in the $(SPOOL) directory.

INVALID_LOG_FILES This macro contains a (comma or space separated) list of files that *condor_preen* considers invalid files to find in the $(LOG) directory. There is no default value.

3.3.16 condor_collector Configuration File Entries

These macros affect the *condor_collector*.

CLASSAD_LIFETIME This macro determines the default maximum age for ClassAds collected by the *condor_collector*. ClassAd older than the maximum age are discarded by the *condor_collector* as stale.

If present, the ClassAd attribute "ClassAdLifetime" specifies the ad's lifetime in seconds. If "ClassAdLifetime" is not present in the ad, the *condor_collector* will use the value of `$(CLASSAD_LIFETIME)`. The macro is defined in terms of seconds, and defaults to 900 (15 minutes).

MASTER_CHECK_INTERVAL This macro defines how often the collector should check for machines that have ClassAds from some daemons, but not from the *condor_master* (*orphaned daemons*) and send e-mail about it. It is defined in seconds and defaults to 10800 (3 hours).

COLLECTOR_REQUIREMENTS A boolean expression that filters out unwanted ClassAd updates. The expression is evaluated for ClassAd updates that have passed through enabled security authorization checks. The default behavior when this expression is not defined is to allow all ClassAd updates to take place. If `False`, a ClassAd update will be rejected.

Stronger security mechanisms are the better way to authorize or deny updates to the *condor_collector*. This configuration variable exists to help those that use host-based security, and do not trust all processes that run on the hosts in the pool. This configuration variable may be used to throw out ClassAds that should not be allowed. For example, for *condor_startd* daemons that run on a fixed port, configure this expression to ensure that only machine ClassAds advertising the expected fixed port are accepted. As a convenience, before evaluating the expression, some basic sanity checks are performed on the ClassAd to ensure that all of the ClassAd attributes used by Condor to contain IP:port information are consistent. To validate this information, the attribute to check is `TARGET.MyAddress`.

CLIENT_TIMEOUT Network timeout that the *condor_collector* uses when talking to any daemons or tools that are sending it a ClassAd update. It is defined in seconds and defaults to 30.

QUERY_TIMEOUT Network timeout when talking to anyone doing a query. It is defined in seconds and defaults to 60.

CONDOR_DEVELOPERS By default, Condor will send e-mail once per week to this address with the output of the *condor_status* command, which lists how many machines are in the pool and how many are running jobs. The default value of condor-admin@cs.wisc.edu will send this report to the Condor Team developers at the University of Wisconsin-Madison. The Condor Team uses these weekly status messages in order to have some idea as to how many Condor pools exist in the world. We appreciate getting the reports, as this is one way we can convince funding agencies that Condor is being used in the real world. If you do not wish this information to be sent to the Condor Team, explicitly set the value to `NONE` to disable this feature, or replace the address with a desired location. If undefined (commented out) in the configuration file, Condor follows its default behavior.

COLLECTOR_NAME This macro is used to specify a short description of your pool. It should be about 20 characters long. For example, the name of the UW-Madison Computer Science Condor Pool is `"UW-Madison CS"`. While this macro might seem similar to `MASTER_NAME` or `SCHEDD_NAME`, it is unrelated. Those settings are used to uniquely identify (and locate)

a specific set of Condor daemons, if there are more than one running on the same machine. The COLLECTOR_NAME setting is just used as a human-readable string to describe the pool, which is included in the updates set to the CONDOR_DEVELOPERS_COLLECTOR (see below).

CONDOR_DEVELOPERS_COLLECTOR By default, every pool sends periodic updates to a central *condor_collector* at UW-Madison with basic information about the status of the pool. Updates include only the number of total machines, the number of jobs submitted, the number of machines running jobs, the host name of the central manager, and the $(COLLECTOR_NAME). These updates help the Condor Team see how Condor is being used around the world. By default, they will be sent to condor.cs.wisc.edu. To discontinue sending updates, explicitly set this macro to NONE. If undefined or commented out in the configuration file, Condor follows its default behavior.

COLLECTOR_UPDATE_INTERVAL This variable is defined in seconds and defaults to 900 (every 15 minutes). It controls the frequency of the periodic updates sent to a central *condor_collector* at UW-Madison as defined by CONDOR_DEVELOPERS_COLLECTOR.

COLLECTOR_SOCKET_BUFSIZE This specifies the buffer size, in bytes, reserved for *condor_collector* network UDP sockets. The default is 10240000, or a ten megabyte buffer. This is a healthy size, even for a large pool. The larger this value, the less likely the *condor_collector* will have stale information about the pool due to dropping update packets. If your pool is small or your central manager has very little RAM, considering setting this parameter to a lower value (perhaps 256000 or 128000).

NOTE: For some Linux distributions, it may be necessary to raise the OS's system-wide limit for network buffer sizes. The parameter that controls this limit is /proc/sys/net/core/rmem_max. You can see the values that the *condor_collector* actually uses by enabling D_FULLDEBUG for the collector and looking at the log line that looks like this:

Reset OS socket buffer size to 2048k (UDP), 255k (TCP).

For changes to this parameter to take effect, *condor_collector* must be restarted.

COLLECTOR_TCP_SOCKET_BUFSIZE This specifies the TCP buffer size, in bytes, reserved for *condor_collector* network sockets. The default is 131072, or a 128 kilobyte buffer. This is a healthy size, even for a large pool. The larger this value, the less likely the *condor_collector* will have stale information about the pool due to dropping update packets. If your pool is small or your central manager has very little RAM, considering setting this parameter to a lower value (perhaps 65536 or 32768).

NOTE: See the note for COLLECTOR_SOCKET_BUFSIZE .

KEEP_POOL_HISTORY This boolean macro is used to decide if the collector will write out statistical information about the pool to history files. The default is False. The location, size, and frequency of history logging is controlled by the other macros.

POOL_HISTORY_DIR This macro sets the name of the directory where the history files reside (if history logging is enabled). The default is the SPOOL directory.

POOL_HISTORY_MAX_STORAGE This macro sets the maximum combined size of the history files. When the size of the history files is close to this limit, the oldest information will be discarded. Thus, the larger this parameter's value is, the larger the time range for which history will be available. The default value is 10000000 (10 Mbytes).

POOL_HISTORY_SAMPLING_INTERVAL This macro sets the interval, in seconds, between samples for history logging purposes. When a sample is taken, the collector goes through the information it holds, and summarizes it. The information is written to the history file once for each 4 samples. The default (and recommended) value is 60 seconds. Setting this macro's value too low will increase the load on the collector, while setting it to high will produce less precise statistical information.

COLLECTOR_DAEMON_STATS A boolean value that controls whether or not the *condor_collector* daemon keeps update statistics on incoming updates. The default value is `True`. If enabled, the *condor_collector* will insert several attributes into the ClassAds that it stores and sends. ClassAds without the `UpdateSequenceNumber` and `DaemonStartTime` attributes will not be counted, and will not have attributes inserted (all modern Condor daemons which publish ClassAds publish these attributes).

The attributes inserted are `UpdatesTotal`, `UpdatesSequenced`, and `UpdatesLost`. `UpdatesTotal` is the total number of updates (of this ClassAd type) the *condor_collector* has received from this host. `UpdatesSequenced` is the number of updates that the *condor_collector* could have as lost. In particular, for the first update from a daemon, it is impossible to tell if any previous ones have been lost or not. `UpdatesLost` is the number of updates that the *condor_collector* has detected as being lost. See page 925 for more information on the added attributes.

COLLECTOR_STATS_SWEEP This value specifies the number of seconds between sweeps of the *condor_collector*'s per-daemon update statistics. Records for daemons which have not reported in this amount of time are purged in order to save memory. The default is two days. It is unlikely that you would ever need to adjust this.

COLLECTOR_DAEMON_HISTORY_SIZE This macro controls the size of the published update history that the Collector inserts into the ClassAds it stores and sends. The default value is 128, which means that history is stored and published for the latest 128 updates. This macro is ignored if `$(COLLECTOR_DAEMON_STATS)` is not enabled.

If this has a non-zero value, the Collector will insert `UpdatesHistory` into the ClassAd (similar to `UpdatesTotal` above). AttrUpdatesHistory is a hexadecimal string which represents a bitmap of the last `COLLECTOR_DAEMON_HISTORY_SIZE` updates. The most significant bit (MSB) of the bitmap represents the most recent update, and the least significant bit (LSB) represents the least recent. A value of zero means that the update was not lost, and a value of 1 indicates that the update was detected as lost.

For example, if the last update was not lost, the previous lost, and the previous two not, the bitmap would be 0100, and the matching hex digit would be "4". Note that the MSB can never be marked as lost because its loss can only be detected by a non-lost update (a "gap" is found in the sequence numbers). Thus, UpdatesHistory = "0x40" would be the history for the last 8 updates. If the next updates are all successful, the values published, after each update, would be: 0x20, 0x10, 0x08, 0x04, 0x02, 0x01, 0x00.

See page 925 for more information on the added attribute.

COLLECTOR_CLASS_HISTORY_SIZE This macro controls the size of the published update history that the Collector inserts into the Collector ClassAds it produces. The default value is zero.

If this has a non-zero value, the Collector will insert "UpdatesClassHistory" into the Collector ClassAd (similar to "UpdatesHistory" above). These are added "per class" of ClassAd, however. The classes refer to the "type" of ClassAds (i.e. "Start"). Additionally, there is a "Total" class created which represents the history of all ClassAds that this Collector receives.

Note that the collector always publishes Lost, Total and Sequenced counts for all ClassAd "classes". This is similar to the statistics gathered if `$(COLLECTOR_DAEMON_STATS)` is enabled.

COLLECTOR_QUERY_WORKERS This macro sets the maximum number of "worker" processes that the Collector can have. When receiving a query request, the UNIX Collector will "fork" a new process to handle the query, freeing the main process to handle other requests. When the number of outstanding "worker" processes reaches this maximum, the request is handled by the main process. This macro is ignored on Windows, and its default value is zero. The default configuration, however, has this set to 16.

COLLECTOR_DEBUG This macro (and other macros related to debug logging in the collector) is described in section 3.3.4 as `<SUBSYS>_DEBUG`.

CONDOR_VIEW_CLASSAD_TYPES Provides the ClassAd types that will be forwarded to the `CONDOR_VIEW_HOST`. The ClassAd types can be found with *condor_status -any*. The default forwarding behavior of the *condor_collector* is equivalent to

```
CONDOR_VIEW_CLASSAD_TYPES=Machine,Submitter
```

There is no default value for this variable.

3.3.17 condor_negotiator Configuration File Entries

These macros affect the *condor_negotiator*.

NEGOTIATOR_INTERVAL Sets how often the *condor_negotiator* starts a negotiation cycle. It is defined in seconds and defaults to 60 (1 minute).

NEGOTIATOR_CYCLE_DELAY An integer value that represents the minimum number of seconds that must pass before a new negotiation cycle may start. The default value is 20. `NEGOTIATOR_CYCLE_DELAY` is intended only for use by Condor experts.

NEGOTIATOR_TIMEOUT Sets the timeout that the negotiator uses on its network connections to the *condor_schedd* and *condor_startd*s. It is defined in seconds and defaults to 30.

NEGOTIATION_CYCLE_STATS_LENGTH Specifies how many recent negotiation cycles should be included in the history that is published in the *condor_negotiator*'s ad. The default is 3 and the maximum allowed value is 100. Setting this value to 0 disables publication of negotiation cycle statistics. The statistics about recent cycles are stored in several attributes per cycle. Each of these attribute names will have a number appended to it to indicate how long ago the cycle happened, for example: `LastNegotiationCycleDuration0`, `LastNegotiationCycleDuration1`, `LastNegotiationCycleDuration2`, ... The attribute numbered 0 applies to the most recent negotiation cycle. The attribute numbered 1 applies to the next most recent negotiation cycle, and so on. See page 923 for a list of attributes that are published.

PRIORITY_HALFLIFE This macro defines the half-life of the user priorities. See section 2.7.2 on User Priorities for details. It is defined in seconds and defaults to 86400 (1 day).

DEFAULT_PRIO_FACTOR This macro sets the priority factor for local users. See section 2.7.2 on User Priorities for details. Defaults to 1.

NICE_USER_PRIO_FACTOR This macro sets the priority factor for nice users. See section 2.7.2 on User Priorities for details. Defaults to 10000000.

REMOTE_PRIO_FACTOR This macro defines the priority factor for remote users (users who who do not belong to the accountant's local domain - see below). See section 2.7.2 on User Priorities for details. Defaults to 10000.

ACCOUNTANT_LOCAL_DOMAIN This macro is used to decide if a user is local or remote. A user is considered to be in the local domain if the UID_DOMAIN matches the value of this macro. Usually, this macro is set to the local UID_DOMAIN. If it is not defined, all users are considered local.

MAX_ACCOUNTANT_DATABASE_SIZE This macro defines the maximum size (in bytes) that the accountant database log file can reach before it is truncated (which re-writes the file in a more compact format). If, after truncating, the file is larger than one half the maximum size specified with this macro, the maximum size will be automatically expanded. The default is 1 megabyte (1000000).

NEGOTIATOR_DISCOUNT_SUSPENDED_RESOURCES This macro tells the negotiator to not count resources that are suspended when calculating the number of resources a user is using. Defaults to false, that is, a user is still charged for a resource even when that resource has suspended the job.

NEGOTIATOR_SOCKET_CACHE_SIZE This macro defines the maximum number of sockets that the *condor_negotiator* keeps in its open socket cache. Caching open sockets makes the negotiation protocol more efficient by eliminating the need for socket connection establishment for each negotiation cycle. The default is currently 16. To be effective, this parameter should be set to a value greater than the number of *condor_schedd*s submitting jobs to the negotiator at any time. If you lower this number, you must run *condor_restart* and not just *condor_reconfig* for the change to take effect.

NEGOTIATOR_INFORM_STARTD Boolean setting that controls if the *condor_negotiator* should inform the *condor_startd* when it has been matched with a job. The default is `True`. When this is set to `False`, the *condor_startd* will never enter the Matched state, and will go directly from Unclaimed to Claimed. Because this notification is done via UDP, if a pool is configured so that the execute hosts do not create UDP command sockets (see the `WANT_UDP_COMMAND_SOCKET` setting described in section 3.3.3 on page 169 for details), the *condor_negotiator* should be configured not to attempt to contact these *condor_startds* by configuring this setting to `False`.

NEGOTIATOR_PRE_JOB_RANK Resources that match a request are first sorted by this expression. If there are any ties in the rank of the top choice, the top resources are sorted by the user-supplied rank in the job ClassAd, then by `NEGOTIATOR_POST_JOB_RANK`, then by `PREEMPTION_RANK` (if the match would cause preemption and there are still any ties in the top choice). `MY` refers to attributes of the machine ClassAd and `TARGET` refers to the job ClassAd. The purpose of the pre job rank is to allow the pool administrator to override any other rankings, in order to optimize overall throughput. For example, it is commonly used to minimize preemption, even if the job rank prefers a machine that is busy. If undefined, this expression has no effect on the ranking of matches. The standard configuration file shipped with Condor specifies an expression to steer jobs away from busy resources:

```
NEGOTIATOR_PRE_JOB_RANK = RemoteOwner =?= UNDEFINED
```

NEGOTIATOR_POST_JOB_RANK Resources that match a request are first sorted by `NEGOTIATOR_PRE_JOB_RANK`. If there are any ties in the rank of the top choice, the top resources are sorted by the user-supplied rank in the job ClassAd, then by `NEGOTIATOR_POST_JOB_RANK`, then by `PREEMPTION_RANK` (if the match would cause preemption and there are still any ties in the top choice). `MY.` refers to attributes of the machine ClassAd and `TARGET.` refers to the job ClassAd. The purpose of the post job rank is to allow the pool administrator to choose between machines that the job ranks equally. The default value is undefined, which causes this rank to have no effect on the ranking of matches. The following example expression steers jobs toward faster machines and tends to fill a cluster of multi-processors by spreading across all machines before filling up individual machines. In this example, the expression is chosen to have no effect when preemption would take place, allowing control to pass on to `PREEMPTION_RANK`.

```
UWCS_NEGOTIATOR_POST_JOB_RANK = \
 (RemoteOwner =?= UNDEFINED) * (KFlops - SlotID)
```

PREEMPTION_REQUIREMENTS When considering user priorities, the negotiator will not preempt a job running on a given machine unless the `PREEMPTION_REQUIREMENTS` expression evaluates to `True` and the owner of the idle job has a better priority than the owner of the running job. The `PREEMPTION_REQUIREMENTS` expression is evaluated within the context of the candidate machine ClassAd and the candidate idle job ClassAd; thus the `MY` scope prefix refers to the machine ClassAd, and the `TARGET` scope prefix refers to the ClassAd of the idle (candidate) job. There is no direct access to the currently running job, but attributes of

the currently running job that need to be accessed in PREEMPTION_REQUIREMENTS can be placed in the machine ClassAd using STARTD_JOB_EXPRS . If not explicitly set in the Condor configuration file, the default value for this expression is True. Note that this setting does not influence other potential causes of preemption, such as startd RANK, or PREEMPT expressions. See section 3.5.9 for a general discussion of limiting preemption.

PREEMPTION_REQUIREMENTS_STABLE A boolean value that defaults to True, implying that all attributes utilized to define the PREEMPTION_REQUIREMENTS variable will not change within a negotiation period time interval. If utilized attributes will change during the negotiation period time interval, then set this variable to False.

PREEMPTION_RANK Resources that match a request are first sorted by NEGOTIATOR_PRE_JOB_RANK. If there are any ties in the rank of the top choice, the top resources are sorted by the user-supplied rank in the job ClassAd, then by NEGOTIATOR_POST_JOB_RANK, then by PREEMPTION_RANK (if the match would cause preemption and there are still any ties in the top choice). MY refers to attributes of the machine ClassAd and TARGET refers to the job ClassAd. This expression is used to rank machines that the job and the other negotiation expressions rank the same. For example, if the job has no preference, it is usually preferable to preempt a job with a small ImageSize instead of a job with a large ImageSize. The default is to rank all preemptable matches the same. However, the negotiator will always prefer to match the job with an idle machine over a preemptable machine, if none of the other ranks express a preference between them.

PREEMPTION_RANK_STABLE A boolean value that defaults to True, implying that all attributes utilized to define the PREEMPTION_RANK variable will not change within a negotiation period time interval. If utilized attributes will change during the negotiation period time interval, then set this variable to False.

NEGOTIATOR_DEBUG This macro (and other settings related to debug logging in the negotiator) is described in section 3.3.4 as <SUBSYS>_DEBUG.

NEGOTIATOR_MAX_TIME_PER_SUBMITTER The maximum number of seconds the *condor_negotiator* will spend with a submitter during one negotiation cycle. Once this time limit has been reached, the *condor_negotiator* will still finish its current pie spin, but it will skip over the submitter if subsequent pie spins are needed to dish out all of the available machines. It defaults to one year. See NEGOTIATOR_MAX_TIME_PER_PIESPIN for more information.

NEGOTIATOR_MAX_TIME_PER_PIESPIN The maximum number of seconds the *condor_negotiator* will spend with a submitter in one pie spin. A negotiation cycle is composed of at least one pie spin, possibly more, depending on whether there are still machines left over after computing fair shares and negotiating with each submitter. By limiting the maximum length of a pie spin or the maximum time per submitter per negotiation cycle, the *condor_negotiator* is protected against spending a long time talking to one submitter, for example someone with a very slow *condor_schedd* daemon. But, this can result in unfair allocation of machines or some machines not being allocated at all. See section 3.4.6 on page 279 for a description of a pie slice.

NEGOTIATOR_MATCH_EXPRS A comma-separated list of macro names that are inserted as ClassAd attributes into matched job ClassAds. The attribute name in the ClassAd will be given the prefix `NegotiatorMatchExpr`, if the macro name does not already begin with that. Example:

```
NegotiatorName = "My Negotiator"
NEGOTIATOR_MATCH_EXPRS = NegotiatorName
```

As a result of the above configuration, jobs that are matched by this *condor_negotiator* will contain the following attribute when they are sent to the *condor_startd*:

```
NegotiatorMatchExprNegotiatorName = "My Negotiator"
```

The expressions inserted by the *condor_negotiator* may be useful in *condor_startd* policy expressions, when the *condor_startd* belongs to multiple Condor pools.

NEGOTIATOR_MATCHLIST_CACHING A boolean value that defaults to `True`. When `True`, it enables an optimization in the *condor_negotiator* that works with auto clustering. In determining the sorted list of machines that a job might use, the job goes to the first machine off the top of the list. If `NEGOTIATOR_MATCHLIST_CACHING` is `True`, and if the next job is part of the same auto cluster, meaning that it is a very similar job, the *condor_negotiator* will reuse the previous list of machines, instead of recreating the list from scratch.

If matching grid resources, and the desire is for a given resource to potentially match multiple times per *condor_negotiator* pass, `NEGOTIATOR_MATCHLIST_CACHING` should be `False`. See section 5.3.10 on page 577 in the subsection on Advertising Grid Resources to Condor for an example.

The following configuration macros affect negotiation for group users.

GROUP_NAMES A comma-separated list of the recognized group names, case insensitive. If undefined (the default), group support is disabled. Group names must not conflict with any user names. That is, if there is a `physics` group, there may not be a `physics` user. Any group that is defined here must also have a quota, or the group will be ignored. Example:

```
GROUP_NAMES = group_physics, group_chemistry
```

GROUP_QUOTA_<groupname> A floating point value to represent a static quota specifying an integral number of machines for the hierarchical group identified by `<groupname>`. It is meaningless to specify a non integer value, since only integral numbers of machines can be allocated. Example:

```
GROUP_QUOTA_group_physics = 20
GROUP_QUOTA_group_chemistry = 10
```

When both static and dynamic quotas are defined for a specific group, the static quota is used and the dynamic quota is ignored.

GROUP_QUOTA_DYNAMIC_<groupname> A floating point value in the range 0.0 to 1.0, inclusive, representing a fraction of a pool's machines (slots) set as a dynamic quota for the hierarchical group identified by <groupname>. For example, the following specifies that a quota of 25% of the total machines are reserved for members of the group_biology group.

```
GROUP_QUOTA_DYNAMIC_group_biology = 0.25
```

The group name must be specified in the GROUP_NAMES list.

This section has not yet been completed

GROUP_PRIO_FACTOR_<groupname> A floating point value greater than or equal to 1.0 to specify the default user priority factor for <groupname>. The group name must also be specified in the GROUP_NAMES list. GROUP_PRIO_FACTOR_<groupname> is evaluated when the negotiator first negotiates for the user as a member of the group. All members of the group inherit the default priority factor when no other value is present. For example, the following setting specifies that all members of the group named group_physics inherit a default user priority factor of 2.0:

```
GROUP_PRIO_FACTOR_group_physics = 2.0
```

GROUP_AUTOREGROUP A boolean value (defaults to False) that when True, causes users who submitted to a specific group to also negotiate a second time with the none group, to be considered with the independent job submitters. This allows group submitted jobs to be matched with idle machines even if the group is over its quota. The user name that is used for accounting and prioritization purposes is still the group user as specified by AccountingGroup in the job ClassAd.

GROUP_AUTOREGROUP_<groupname> This is the same as GROUP_AUTOREGROUP, but it is settable on a per-group basis. If no value is specified for a given group, the default behavior is determined by GROUP_AUTOREGROUP, which in turn defaults to False.

NEGOTIATOR_CONSIDER_PREEMPTION For expert users only. A boolean value (defaults to True), that when False, can cause the negotiator to run faster and also have better spinning pie accuracy. *Only set this to* `False` *if* `PREEMPTION_REQUIREMENTS` *is* `False`*, and if all condor_startd rank expressions are* `False`.

STARTD_AD_REEVAL_EXPR A boolean value evaluated in the context of each machine ClassAd within a negotiation cycle that determines whether the ClassAd from the *condor_collector* is to replace the stashed ClassAd utilized during the previous negotiation cycle. When True, the ClassAd from the *condor_collector* does replace the stashed one. When not defined, the default value is to replace the stashed ClassAd if the stashed ClassAd's sequence number is older than its potential replacement.

3.3.18 condor_procd Configuration File Macros

USE_PROCD This boolean parameter is used to determine whether the *condor_procd* will be used for managing process families. If the *condor_procd* is not used, each daemon will run the process family tracking logic on its own. Use of the *condor_procd* results in improved scalability because only one instance of this logic is required. The *condor_procd* is required when using privilege separation (see Section 3.6.14) or group ID-based process tracking (see Section 3.13.12). In either of these cases, the USE_PROCD setting will be ignored and a *condor_procd* will always be used. By default, the *condor_master* will not use a *condor_procd* but all other daemons that need process family tracking will. A daemon that uses the *condor_procd* will start a *condor_procd* for use by itself and all of its child daemons.

PROCD_MAX_SNAPSHOT_INTERVAL This setting determines the maximum time that the *condor_procd* will wait between probes of the system for information about the process families it is tracking.

PROCD_LOG Specifies a log file for the *condor_procd* to use. Note that by design, the *condor_procd* does not include most of the other logic that is shared amongst the various Condor daemons. This is because the *condor_procd* is a component of the PrivSep Kernel (see Section 3.6.14 for more information regarding privilege separation). This means that the *condor_procd* does not include the normal Condor logging subsystem, and thus multiple debug levels are not supported. PROCD_LOG is not set by default and is only intended to debug problems should they arise. Note, however, that enabling D_PROCFAMILY in the debug level for any other daemon will cause it to log all interactions with the *condor_procd*.

MAX_PROCD_LOG Controls the maximum length in bytes to which the *condor_procd* log will be allowed to grow. The log file will grow to the specified length, then be saved to a file with the suffix .old. The .old file is overwritten each time the log is saved, thus the maximum space devoted to logging will be twice the maximum length of this log file. A value of 0 specifies that the file may grow without bounds. The default is 10 Mbyte.

PROCD_ADDRESS This specifies the address that the *condor_procd* will use to receive requests from other Condor daemons. On Unix, this should point to a file system location that can be used for a named pipe. On Windows, named pipes are also used but they do not exist in the file system. The default setting therefore depends on the platform: $(LOCK)/procd_pipe on Unix and \\.\pipe\procd_pipe on Windows.

USE_GID_PROCESS_TRACKING A boolean value that defaults to False. When True, a job's initial process is assigned a dedicated GID which is further used by the *condor_procd* to reliably track all processes associated with a job. When True, values for MIN_TRACKING_GID and MAX_TRACKING_GID must also be set, or Condor will abort, logging an error message. See section 3.13.12 on page 464 for a detailed description.

MIN_TRACKING_GID An integer value, that together with MAX_TRACKING_GID specify a range of GIDs to be assigned on a per slot basis for use by the *condor_procd* in tracking processes associated with a job. See section 3.13.12 on page 464 for a detailed description.

MAX_TRACKING_GID An integer value, that together with MIN_TRACKING_GID specify a range of GIDs to be assigned on a per slot basis for use by the *condor_procd* in tracking processes associated with a job. See section 3.13.12 on page 464 for a detailed description.

3.3.19 condor_credd Configuration File Macros

These macros affect the *condor_credd*.

CREDD_HOST The host name of the machine running the *condor_credd* daemon.

CREDD_CACHE_LOCALLY A boolean value that defaults to False. When True, the first successful password fetch operation to the *condor_credd* daemon causes the password to be stashed in a local, secure password store. Subsequent uses of that password do not require communication with the *condor_credd* daemon.

3.3.20 condor_gridmanager Configuration File Entries

These macros affect the *condor_gridmanager*.

GRIDMANAGER_LOG Defines the path and file name for the log of the *condor_gridmanager*. The owner of the file is the condor user.

GRIDMANAGER_CHECKPROXY_INTERVAL The number of seconds between checks for an updated X509 proxy credential. The default is 10 minutes (600 seconds).

GRIDMANAGER_MINIMUM_PROXY_TIME The minimum number of seconds before expiration of the X509 proxy credential for the gridmanager to continue operation. If seconds until expiration is less than this number, the gridmanager will shutdown and wait for a refreshed proxy credential. The default is 3 minutes (180 seconds).

HOLD_JOB_IF_CREDENTIAL_EXPIRES True or False. Defaults to True. If True, and for grid universe jobs only, Condor-G will place a job on hold GRIDMANAGER_MINIMUM_PROXY_TIME seconds before the proxy expires. If False, the job will stay in the last known state, and Condor-G will periodically check to see if the job's proxy has been refreshed, at which point management of the job will resume.

GRIDMANAGER_CONTACT_SCHEDD_DELAY The minimum number of seconds between connections to the *condor_schedd*. The default is 5 seconds.

GRIDMANAGER_JOB_PROBE_INTERVAL The number of seconds between active probes of the status of a submitted job. The default is 5 minutes (300 seconds).

CONDOR_JOB_POLL_INTERVAL After a condor grid type job is submitted, how often (in seconds) the *condor_gridmanager* should probe the remote *condor_schedd* to check the jobs status. This defaults to 300 seconds (5 minutes). Setting this to a lower number will decrease

latency (Condor will discover that a job has finished more quickly), but will increase network traffic.

GRIDMANAGER_RESOURCE_PROBE_INTERVAL When a resource appears to be down, how often (in seconds) the *condor_gridmanager* should ping it to test if it is up again.

GRIDMANAGER_RESOURCE_PROBE_DELAY The number of seconds between pings of a remote resource that is currently down. The default is 5 minutes (300 seconds).

GRIDMANAGER_EMPTY_RESOURCE_DELAY The number of seconds that the *condor_gridmanager* retains information about a grid resource, once the *condor_gridmanager* has no active jobs on that resource. An active job is a grid universe job that is in the queue, but is not in the HELD state. Defaults to 300 seconds.

GRIDMANAGER_MAX_SUBMITTED_JOBS_PER_RESOURCE An integer value that limits the number of jobs that a *condor_gridmanager* daemon will submit to a resource. It is useful for controlling the number of *jobmanager* processes running on the front-end node of a cluster. This number may be exceeded, if it is reduced through the use of *condor_reconfig* while the *condor_gridmanager* is running, or if the *condor_gridmanager* receives new jobs from the *condor_schedd* that were already submitted (that is, their `GridJobId` is not undefined). In these cases, submitted jobs will not be killed, but no new jobs can be submitted until the number of submitted jobs falls below the current limit. Defaults to 1000.

GRIDMANAGER_MAX_JOBMANAGERS_PER_RESOURCE For grid jobs of type **gt2**, limits the number of globus-job-manager processes that the *condor_gridmanager* lets run at a time on the remote head node. Allowing too many globus-job-managers to run causes severe load on the headnote, possibly making it non-functional. This number may be exceeded if it is reduced through the use of *condor_reconfig* while the *condor_gridmanager* is running or if some globus-job-managers take a few extra seconds to exit. The value 0 means there is no limit. The default value is 10.

GRIDMANAGER_MAX_WS_DESTROYS_PER_RESOURCE For grid jobs of type **gt4**, limits the number of destroy commands that the *condor_gridmanager* will issue at a time to each WS GRAM server. Too many destroy commands can have severe effects on the server. The default value is 5.

GAHP The full path to the binary of the GAHP server. This configuration variable is no longer used. Use `GT2_GAHP` at section 3.3.20 instead.

GAHP_ARGS Arguments to be passed to the GAHP server. This configuration variable is no longer used.

GRIDMANAGER_GAHP_CALL_TIMEOUT The number of seconds after which a pending GAHP command should time out. The default is 5 minutes (300 seconds).

GRIDMANAGER_MAX_PENDING_REQUESTS The maximum number of GAHP commands that can be pending at any time. The default is 50.

GRIDMANAGER_CONNECT_FAILURE_RETRY_COUNT The number of times to retry a command that failed due to a timeout or a failed connection. The default is 3.

GRIDMANAGER_GLOBUS_COMMIT_TIMEOUT The duration, in seconds, of the two phase commit timeout to Globus for gt2 jobs only. This maps directly to the `two_phase` setting in the Globus RSL.

GLOBUS_GATEKEEPER_TIMEOUT The number of seconds after which if a gt2 grid universe job fails to ping the gatekeeper, the job will be put on hold. Defaults to 5 days (in seconds).

GRAM_VERSION_DETECTION A boolean value that defaults to `True`. When `True`, the *condor_gridmanager* treats grid types `gt2` and `gt5` identically, and queries each server to determine which protocol it is using. When `False`, the *condor_gridmanager* trusts the grid type provided in job attribute `GridResource`, and treats the server accordingly. Beware that identifying a `gt2` server as `gt5` can result in overloading the server, if a large number of jobs are submitted.

GRIDFTP_URL_BASE Specifies an existing *GridFTP* server on the local system to be used for file transfers for gt4 grid universe jobs. The value is given as the base of a URL, such as `gsiftp://mycomp.foo.edu:2118`. The default is for Condor to launch temporary *GridFTP* servers as needed for file transfer.

C_GAHP_LOG The complete path and file name of the Condor GAHP server's log. There is no default value. The expected location as defined in the example configuration is `/temp/CGAHPLog.$(USERNAME)`.

MAX_C_GAHP_LOG The maximum size of the `C_GAHP_LOG`.

C_GAHP_WORKER_THREAD_LOG The complete path and file name of the Condor GAHP worker process' log. There is no default value. The expected location as defined in the example configuration is `/temp/CGAHPWorkerLog.$(USERNAME)`.

C_GAHP_CONTACT_SCHEDD_DELAY The number of seconds that the *condor_C-gahp* daemon waits between consecutive connections to the remote *condor_schedd* in order to send batched sets of commands to be executed on that remote *condor_schedd* daemon. The default value is 5.

GLITE_LOCATION The complete path to the directory containing the Glite software. There is no default value. The expected location as given in the example configuration is `$(LIB)/glite`. The necessary Glite software is included with Condor, and is required for pbs and lsf jobs.

AMAZON_EC2_URL The URL Condor should use when contacting the Amazon EC2 service. This URL may include a user name and password as in the implied syntax example:

```
protocol://username:password@domain:port/path/to/resource
```

The default value is `https://ec2.amazonaws.com/`. This parameter is only used for queued jobs when upgrading to Condor 7.5.4 or beyond.

AMAZON_HTTP_PROXY The http proxy that Condor should use when contacting the Amazon EC2 service. The default is to not use a proxy.

CONDOR_GAHP The complete path and file name of the Condor GAHP executable. There is no default value. The expected location as given in the example configuration is `$(SBIN)/condor_c-gahp`.

AMAZON_GAHP The complete path and file name of the Amazon GAHP executable. There is no default value. The expected location as given in the example configuration is `$(SBIN)/amazon_gahp`.

GT2_GAHP The complete path and file name of the GT2 GAHP executable. There is no default value. The expected location as given in the example configuration is `$(SBIN)/gahp_server`.

GT4_GAHP The complete path and file name of the wrapper script that invokes the GT4 GAHP executable. There is no default value. The expected location as given in the example configuration is `$(SBIN)/gt4_gahp`.

PBS_GAHP The complete path and file name of the PBS GAHP executable. There is no default value. The expected location as given in the example configuration is `$(GLITE_LOCATION)/bin/batch_gahp`.

LSF_GAHP The complete path and file name of the LSF GAHP executable. There is no default value. The expected location as given in the example configuration is `$(GLITE_LOCATION)/bin/batch_gahp`.

UNICORE_GAHP The complete path and file name of the wrapper script that invokes the Unicore GAHP executable. There is no default value. The expected location as given in the example configuration is `$(SBIN)/unicore_gahp`.

NORDUGRID_GAHP The complete path and file name of the wrapper script that invokes the NorduGrid GAHP executable. There is no default value. The expected location as given in the example configuration is `$(SBIN)/nordugrid_gahp`.

CREAM_GAHP The complete path and file name of the CREAM GAHP executable. There is no default value. The expected location as given in the example configuration is `$(SBIN)/cream_gahp`.

DELTACLOUD_GAHP The complete path and file name of the Deltacloud GAHP executable. There is no default value. The expected location as given in the example configuration is `$(SBIN)/deltacloud_gahp`.

3.3.21 condor_job_router Configuration File Entries

These macros affect the *condor_job_router* daemon.

JOB_ROUTER_DEFAULTS Defined by a single ClassAd in New ClassAd syntax, used to provide default values for all routes in the *condor_job_router* daemon's routing table. Where an attribute is set outside of these defaults, that attribute value takes precedence.

JOB_ROUTER_ENTRIES Specification of the job routing table. It is a list of ClassAds, in New ClassAd syntax, where each individual ClassAd is surrounded by square brackets, and the ClassAds are separated from each other by spaces. Each ClassAd describes one entry in the routing table, and each describes a site that jobs may be routed to.

A *condor_reconfig* command causes the *condor_job_router* daemon to rebuild the routing table. Routes are distinguished by a routing table entry's ClassAd attribute `Name`. Therefore, a `Name` change in an existing route has the potential to cause the inaccurate reporting of routes.

Instead of setting job routes using this configuration variable, they may be read from an external source using the `JOB_ROUTER_ENTRIES_FILE` or be dynamically generated by an external program via the `JOB_ROUTER_ENTRIES_CMD` configuration variable.

JOB_ROUTER_ENTRIES_FILE A path and file name of a file that contains the ClassAds, in New ClassAd syntax, describing the routing table. The specified file is periodically reread to check for new information. This occurs every `$(JOB_ROUTER_ENTRIES_REFRESH)` seconds.

JOB_ROUTER_ENTRIES_CMD Specifies the command line of an external program to run. The output of the program defines or updates the routing table, and the output must be given in New ClassAd syntax. The specified command is periodically rerun to regenerate or update the routing table. This occurs every `$(JOB_ROUTER_ENTRIES_REFRESH)` seconds. Specify the full path and file name of the executable within this command line, as no assumptions may be made about the current working directory upon command invocation. To enter spaces in any command-line arguments or in the command name itself, surround the right hand side of this definition with double quotes, and use single quotes around individual arguments that contain spaces. This is the same as when dealing with spaces within job arguments in a Condor submit description file.

JOB_ROUTER_ENTRIES_REFRESH The number of seconds between updates to the routing table described by `JOB_ROUTER_ENTRIES_FILE` or `JOB_ROUTER_ENTRIES_CMD`. The default value is 0, meaning no periodic updates occur. With the default value of 0, the routing table can be modified when a *condor_reconfig* command is invoked or when the *condor_job_router* daemon restarts.

JOB_ROUTER_LOCK This specifies the name of a lock file that is used to ensure that multiple instances of condor_job_router never run with the same `JOB_ROUTER_NAME`. Multiple instances running with the same name could lead to mismanagement of routed jobs. The default value is `$(LOCK)/$(JOB_ROUTER_NAME)Lock`.

JOB_ROUTER_SOURCE_JOB_CONSTRAINT Specifies a global `Requirements` expression that must be true for all newly routed jobs, in addition to any `Requirements` specified within a routing table entry. In addition to the configurable constraints, the *condor_job_router* also has some hard-coded constraints. It avoids recursively routing jobs by requiring that the job's attribute `RoutedBy` does not match `JOB_ROUTER_NAME` . When not running as root, it also avoids routing jobs belonging to other users.

JOB_ROUTER_MAX_JOBS An integer value representing the maximum number of jobs that may be routed, summed over all routes. The default value is -1, which means an unlimited number of jobs may be routed.

MAX_JOB_MIRROR_UPDATE_LAG An integer value that administrators will rarely consider changing, representing the maximum number of seconds the *condor_job_router* daemon waits, before it decides that routed copies have gone awry, due to the failure of events to appear in the *condor_schedd*'s job queue log file. The default value is 600. As the *condor_job_router* daemon uses the *condor_schedd*'s job queue log file entries for synchronization of routed copies, when an expected log file event fails to appear after this wait period, the *condor_job_router* daemon acts presuming the expected event will never occur.

JOB_ROUTER_POLLING_PERIOD An integer value representing the number of seconds between cycles in the *condor_job_router* daemon's task loop. The default is 10 seconds. A small value makes the *condor_job_router* daemon quick to see new candidate jobs for routing. A large value makes the *condor_job_router* daemon generate less overhead at the cost of being slower to see new candidates for routing. For very large job queues where a few minutes of routing latency is no problem, increasing this value to a few hundred seconds would be reasonable.

JOB_ROUTER_NAME A unique identifier utilized to name multiple instances of the *condor_job_router* daemon on the same machine. Each instance must have a different name, or all but the first to start up will refuse to run. The default is "jobrouter".

Changing this value when routed jobs already exist is not currently gracefully handled. However, it can be done if one also uses *condor_qedit* to change the value of `ManagedManager` and `RoutedBy` from the old name to the new name. The following commands may be helpful:

```
condor_qedit -constraint 'RoutedToJobId =!= undefined && \
  ManagedManager == "insert_old_name"' \
  ManagedManager '"insert_new_name"'
condor_qedit -constraint 'RoutedBy == "insert_old_name"' \
  RoutedBy '"insert_new_name"'
```

JOB_ROUTER_RELEASE_ON_HOLD A boolean value that defaults to `True`. It controls how the *condor_job_router* handles the routed copy when it goes on hold. When `True`, the *condor_job_router* leaves the original job ClassAd in the same state as when claimed. When `False`, the *condor_job_router* does not attempt to reset the original job ClassAd to a pre-claimed state upon yielding control of the job.

3.3.22 condor_lease_manager Configuration File Entries

These macros affect the *condor_lease_manager*.

The *condor_lease_manager* expects to use the syntax

```
<subsystem name>.<parameter name>
```

in configuration. This allows multiple instances of the *condor_lease_manager* to be easily configured using the syntax

```
<subsystem name>.<local name>.<parameter name>
```

LeaseManager.GETADS_INTERVAL An integer value, given in seconds, that controls the frequency with which the *condor_lease_manager* pulls relevant resource ClassAds from the *condor_collector*. The default value is 60 seconds, with a minimum value of 2 seconds.

LeaseManager.UPDATE_INTERVAL An integer value, given in seconds, that controls the frequency with which the *condor_lease_manager* sends its ClassAds to the *condor_collector*. The default value is 60 seconds, with a minimum value of 5 seconds.

LeaseManager.PRUNE_INTERVAL An integer value, given in seconds, that controls the frequency with which the *condor_lease_manager prunes* its leases. This involves checking all leases to see if they have expired. The default value is 60 seconds, with no minimum value.

LeaseManager.DEBUG_ADS A boolean value that defaults to `False`. When `True`, it enables extra debugging information about the resource ClassAds that it retrieves from the *condor_collector* and about the search ClassAds that it sends to the *condor_collector*.

LeaseManager.MAX_LEASE_DURATION An integer value representing seconds which determines the maximum duration of a lease. This can be used to provide a hard limit on lease durations. Normally, the *condor_lease_manager* honors the `MaxLeaseDuration` attribute from the resource ClassAd. If this configuration variable is defined, it limits the effective maximum duration for all resources to this value. The default value is 1800 seconds.

Note that leases can be renewed, and thus can be extended beyond this limit. To provide a limit on the total duration of a lease, use `LeaseManager.MAX_TOTAL_LEASE_DURATION`.

LeaseManager.MAX_TOTAL_LEASE_DURATION An integer value representing seconds used to limit the *total* duration of leases, over all its renewals. The default value is 3600 seconds.

LeaseManager.DEFAULT_MAX_LEASE_DURATION The *condor_lease_manager* uses the `MaxLeaseDuration` attribute from the resource ClassAd to limit the lease duration. If this attribute is not present in a resource ClassAd, then this configuration variable is used instead. This integer value is given in units of seconds, with a default value of 60 seconds.

LeaseManager.CLASSAD_LOG This variable defines a full path and file name to the location where the *condor_lease_manager* keeps persistent state information. This variable has no default value.

LeaseManager.QUERY_ADTYPE This parameter controls the type of the query in the ClassAd sent to the *condor_collector*, which will control the types of ClassAds returned by the *condor_collector*. This parameter must be a valid ClassAd type name, with a default value of `"Any"`.

LeaseManager.QUERY_CONSTRAINTS A ClassAd expression that controls the constraint in the query sent to the *condor_collector*. It is used to further constrain the types of ClassAds from the *condor_collector*. There is no default value, resulting in no constraints being placed on query.

3.3.23 condor_hdfs Configuration File Entries

These macros affect the *condor_hdfs* daemon. Many of these variables determine how the *condor_hdfs* daemon sets the HDFS XML configuration.

HDFS_HOME The directory path for the Hadoop file system installation directory. Defaults to `$(RELEASE_DIR)/libexec`. This directory is required to contain

- directory `lib`, containing all necessary jar files for the execution of a Name node and Data nodes.
- directory `conf`, containing default Hadoop file system configuration files with names that conform to `*-site.xml`.
- directory `webapps`, containing JavaServer pages (jsp) files for the Hadoop file system's embedded server.

HDFS_NAMENODE The host and port number for the HDFS Name node. There is no default value for this required variable. Defines the value of `fs.default.name` in the HDFS XML configuration.

HDFS_NAMENODE_WEB The IP address and port number for the HDFS embedded web server within the Name node with the syntax of `a.b.c.d:portnumber`. There is no default value for this required variable. Defines the value of `dfs.http.address` in the HDFS XML configuration.

HDFS_DATANODE_WEB The IP address and port number for the HDFS embedded web server within the Name node with the syntax of `a.b.c.d:portnumber`. The default value for this optional variable is 0.0.0.0:0, which means bind to the default interface on a dynamic port. Defines the value of `dfs.datanode.http.address` in the HDFS XML configuration.

HDFS_NAMENODE_DIR The path to the directory on a local file system where the Name node will store its meta-data for file blocks. There is no default value for this variable; it is required to be defined for the Name node machine. Defines the value of `dfs.name.dir` in the HDFS XML configuration.

HDFS_DATANODE_DIR The path to the directory on a local file system where the Data node will store file blocks. There is no default value for this variable; it is required to be defined for a Data node machine. Defines the value of `dfs.data.dir` in the HDFS XML configuration.

HDFS_DATANODE_ADDRESS The IP address and port number of this machine's Data node. There is no default value for this variable; it is required to be defined for a Data node machine, and may be given the value `0.0.0.0:0` as a Data node need not be running on a known port. Defines the value of `dfs.datanode.address` in the HDFS XML configuration.

HDFS_NODETYPE A host name, to identify the machine acting as the Name node.

HDFS_BACKUPNODE The host address and port number for the HDFS Backup node. There is no default value. It defines the value of the HDFS dfs.namenode.backup.address field in the HDFS XML configuration file.

HDFS_BACKUPNODE_WEB The address and port number for the HDFS embedded web server within the Backup node, with the syntax of hdfs://<host_address>:<portnumber>. There is no default value for this required variable. It defines the value of dfs.namenode.backup.http-address in the HDFS XML configuration.

HDFS_NAMENODE_ROLE If this machine is selected to be the Name node, then by a role should defined. Defined values are `ACTIVE`, `BACKUP`, `CHECKPOINT`, and `STANDBY`. The default value is `ACTIVE`. The `STANDBY` value exists for future expansion. If `HDFS_NODETYPE` is selected to be Data node (`HDFS_DATANODE`), then this variable is ignored.

HDFS_LOG4J Used to set the configuration for the HDFS debugging level. Currently one of `OFF`, `FATAL`, `ERROR`, `WARN`, `INFODEBUG`, `ALL` or `INFO`. Debugging output is written to `$(LOG)/hdfs.log`. The default value is `INFO`.

HDFS_ALLOW A comma separated list of hosts that are authorized with read and write access to the invoked HDFS. Note that this configuration variable name is likely to change to `HOSTALLOW_HDFS`.

HDFS_DENY A comma separated list of hosts that are denied access to the invoked HDFS. Note that this configuration variable name is likely to change to `HOSTDENY_HDFS`.

HDFS_NAMENODE_CLASS An optional value that specifies the class to invoke. The default value is `org.apache.hadoop.hdfs.server.namenode.NameNode`.

HDFS_DATANODE_CLASS An optional value that specifies the class to invoke. The default value is `org.apache.hadoop.hdfs.server.datanode.DataNode`.

HDFS_SITE_FILE The optional value that specifies the HDFS XML configuration file to generate. The default value is `hdfs-site.xml`.

HDFS_REPLICATION An integer value that facilitates setting the replication factor of an HDFS, defining the value of `dfs.replication` in the HDFS XML configuration. This configuration variable is optional, as the HDFS has its own default value of 3 when not set through configuration.

3.3.24 Grid Monitor Configuration File Entries

These macros affect the Grid Monitor.

ENABLE_GRID_MONITOR A boolean value that when `True` enables the Grid Monitor. The Grid Monitor is used to reduce load on Globus gatekeepers. This parameter only affects grid jobs of type **gt2**. The variable `GRID_MONITOR` must also be correctly configured. Defaults to `True`. See section 5.3.2 on page 570 for more information.

GRID_MONITOR The complete path name of the *grid_monitor.sh* tool used to reduce the load on Globus gatekeepers. This parameter only affects grid jobs of type **gt2**. This parameter is not referenced unless ENABLE_GRID_MONITOR is set to True (the default value).

GRID_MONITOR_HEARTBEAT_TIMEOUT The integer number of seconds that may pass without hearing from a working Grid Monitor before it is assumed to be dead. Defaults to 300 (5 minutes). Increasing this number will improve the ability of the Grid Monitor to survive in the face of transient problems, but will also increase the time before Condor notices a problem.

GRID_MONITOR_RETRY_DURATION When Condor-G attempts to start the Grid Monitor at a particular site, it will wait this many seconds to start hearing from the Grid Monitor. Defaults to 900 (15 minutes). If this duration passes without success, the Grid Monitor will be disabled for the site in question for the period of time set by GRID_MONITOR_DISABLE_TIME.

GRID_MONITOR_NO_STATUS_TIMEOUT Jobs can disappear from the Grid Monitor's status reports for short periods of time under normal circumstances, but a prolonged absence is often a sign of problems on the remote machine. This variable sets the amount of time (in seconds) that a job can be absent before the *condor_gridmanager* reacts by restarting the GRAM *jobmanager*. The default is 900, which is 15 minutes.

GRID_MONITOR_DISABLE_TIME When an error occurs with a Grid Monitor job, this parameter controls how long the *condor_gridmanager* will wait before attempting to start a new Grid Monitor job. The value is in seconds and the default is 3600 (1 hour).

3.3.25 Configuration File Entries Relating to Grid Usage and Glidein

These macros affect the Condor's usage of grid resources and glidein.

GLIDEIN_SERVER_URLS A comma or space-separated list of URLs that contain the binaries that must be copied by *condor_glidein*. There are no default values, but working URLs that copy from the UW site are provided in the distributed sample configuration files.

GLEXEC_JOB A boolean value that defaults to False. When True, it enables the use of *glexec* on the machine.

GLEXEC The full path and file name of the *glexec* executable.

3.3.26 Configuration File Entries for DAGMan

These macros affect the operation of DAGMan and DAGMan jobs within Condor.

Note: Many, if not all, of these configuration variables will be most appropriately set on a per DAG basis, rather than in the global Condor configuration files. Per DAG configuration is explained in section 2.10.6.

DAGMAN_USER_LOG_SCAN_INTERVAL An integer value representing the number of seconds that *condor_dagman* waits between checking job log files for status updates. Setting this value lower than the default increases the CPU time *condor_dagman* spends checking files, perhaps fruitlessly, but increases responsiveness to nodes completing or failing. The legal range of values is 1 to INT_MAX. If not defined, it defaults to 5 seconds.

DAGMAN_DEBUG_CACHE_ENABLE A boolean value that determines if log line caching for the `dagman.out` file should be enabled in the *condor_dagman* process to increase performance (potentially by orders of magnitude) when writing the `dagman.out` file to an NFS server. Currently, this cache is only utilized in Recovery Mode. If not defined, it defaults to `False`.

DAGMAN_DEBUG_CACHE_SIZE An integer value representing the number of bytes of log lines to be stored in the log line cache. When the cache surpasses this number, the entries are written out in one call to the logging subsystem. A value of zero is not recommended since each log line would surpass the cache size and be emitted in addition to bracketing log lines explaining that the flushing was happening. The legal range of values is 0 to INT_MAX. If defined with a value less than 0, the value 0 will be used. If not defined, it defaults to 5 Megabytes.

DAGMAN_MAX_SUBMITS_PER_INTERVAL An integer that controls how many individual jobs *condor_dagman* will submit in a row before servicing other requests (such as a *condor_rm*). The legal range of values is 1 to 1000. If defined with a value less than 1, the value 1 will be used. If defined with a value greater than 1000, the value 1000 will be used. If not defined, it defaults to 5.

DAGMAN_MAX_SUBMIT_ATTEMPTS An integer that controls how many times in a row *condor_dagman* will attempt to execute *condor_submit* for a given job before giving up. Note that consecutive attempts use an exponential backoff, starting with 1 second. The legal range of values is 1 to 16. If defined with a value less than 1, the value 1 will be used. If defined with a value greater than 16, the value 16 will be used. Note that a value of 16 would result in *condor_dagman* trying for approximately 36 hours before giving up. If not defined, it defaults to 6 (approximately two minutes before giving up).

DAGMAN_SUBMIT_DELAY An integer that controls the number of seconds that *condor_dagman* will sleep before submitting consecutive jobs. It can be increased to help reduce the load on the *condor_schedd* daemon. The legal range of values is 0 to 60. If defined with a value less than 0, the value 0 will be used. If defined with a value greater than 60, the value 60 will be used. The default value is 0.

DAGMAN_STARTUP_CYCLE_DETECT A boolean value that defaults to `False`. When `True`, causes *condor_dagman* to check for cycles in the DAG before submitting DAG node jobs, in addition to its run time cycle detection.

DAGMAN_RETRY_SUBMIT_FIRST A boolean value that controls whether a failed submit is retried first (before any other submits) or last (after all other ready jobs are submitted). If this value is set to `True`, when a job submit fails, the job is placed at the head of the queue of ready jobs, so that it will be submitted again before any other jobs are submitted. This had been the behavior of *condor_dagman*. If this value is set to `False`, when a job submit fails, the job is placed at the tail of the queue of ready jobs. If not defined, it defaults to `True`.

DAGMAN_RETRY_NODE_FIRST A boolean value that controls whether a failed node with retries is retried first (before any other ready nodes) or last (after all other ready nodes). If this value is set to `True`, when a node with retries fails after the submit succeeded, the node is placed at the head of the queue of ready nodes, so that it will be tried again before any other jobs are submitted. If this value is set to `False`, when a node with retries fails, the node is placed at the tail of the queue of ready nodes. This had been the behavior of *condor_dagman*. If not defined, it defaults to `False`.

DAGMAN_MAX_JOBS_IDLE An integer value that controls the maximum number of idle node jobs allowed within the DAG before *condor_dagman* temporarily stops submitting jobs. Once idle jobs start to run, *condor_dagman* will resume submitting jobs. If both the command line option and the configuration parameter are specified, the command line option overrides the configuration variable. Unfortunately, `DAGMAN_MAX_JOBS_IDLE` currently counts each individual process within a cluster as a job, which is inconsistent with `DAGMAN_MAX_JOBS_SUBMITTED`. The default is that there is no limit on the maximum number of idle jobs.

DAGMAN_MAX_JOBS_SUBMITTED An integer value that controls the maximum number of node jobs within the DAG that will be submitted to Condor at one time. Note that this variable has the same functionality as the **-maxjobs** *c*ommand line option to *condor_submit_dag*. If both the command line option and the configuration parameter are specified, the command line option overrides the configuration variable. A single invocation of *condor_submit* counts as one job, even if the submit file produces a multi-job cluster. The default is that there is no limit on the maximum number of jobs run at one time.

DAGMAN_MUNGE_NODE_NAMES A boolean value that controls whether *condor_dagman* automatically renames nodes when running multiple DAGs. The renaming is done to avoid possible name conflicts. If this value is set to `True`, all node names have the DAG number followed by the period character (`.`) prepended to them. For example, the first DAG specified on the *condor_submit_dag* command line is considered DAG number 0, the second is DAG number 1, etc. So if DAG number 2 has a node named B, that node will internally be renamed to 2.B. If not defined, `DAGMAN_MUNGE_NODE_NAMES` defaults to `True`.

DAGMAN_IGNORE_DUPLICATE_JOB_EXECUTION This configuration variable is no longer used. The improved functionality of the `DAGMAN_ALLOW_EVENTS` macro eliminates the need for this variable.

For completeness, here is the definition for historical purposes: A boolean value that controls whether *condor_dagman* aborts or continues with a DAG in the rare case that Condor erroneously executes the job within a DAG node more than once. A bug in Condor very occasionally causes a job to run twice. Running a job twice is contrary to the semantics of a DAG. The configuration macro `DAGMAN_IGNORE_DUPLICATE_JOB_EXECUTION` determines whether *condor_dagman* considers this a fatal error or not. The default value is `False`; *condor_dagman* considers running the job more than once a fatal error, logs this fact, and aborts the DAG. When set to `True`, *condor_dagman* still logs this fact, but continues with the DAG.

This configuration macro is to remain at its default value except in the case where a site encounters the Condor bug in which DAG job nodes are executed twice, and where it is certain that having a DAG job node run twice will not corrupt the DAG. The logged messages within

`*.dagman.out` files in the case of that a node job runs twice contain the string "EVENT ERROR."

DAGMAN_ALLOW_EVENTS An integer that controls which bad events are considered fatal errors by *condor_dagman*. This macro replaces and expands upon the functionality of the `DAGMAN_IGNORE_DUPLICATE_JOB_EXECUTION` macro. If `DAGMAN_ALLOW_EVENTS` is set, it overrides the setting of `DAGMAN_IGNORE_DUPLICATE_JOB_EXECUTION`.

The `DAGMAN_ALLOW_EVENTS` value is a logical bitwise OR of the following values:

0 = allow no bad events

1 = allow all bad events, *except* the event `"job re-run after terminated event"`

2 = allow terminated/aborted event combination

4 = allow a `"job re-run after terminated event"` bug

8 = allow garbage or orphan events

16 = allow an execute or terminate event before job's submit event

32 = allow two terminated events per job, as sometimes seen with grid jobs

64 = allow duplicated events in general

The default value is 114, which allows terminated/aborted event combination, allows an execute and/or terminated event before job's submit event, allows double terminated events, and allows general duplicate events.

As examples, a value of 6 instructs *condor_dagman* to allow both the terminated/aborted event combination and the `"job re-run after terminated event"` bug. A value of 0 means that any bad event will be considered a fatal error.

A value of 5 will never abort the DAG because of a bad event. But this value should almost never be used, because the `"job re-run after terminated event"` bug breaks the semantics of the DAG.

DAGMAN_DEBUG This variable is described in section 3.3.4 as `<SUBSYS>_DEBUG`.

MAX_DAGMAN_LOG This variable is described in section 3.3.4 as `MAX_<SUBSYS>_LOG`.

DAGMAN_CONDOR_SUBMIT_EXE The executable that *condor_dagman* will use to submit Condor jobs. If not defined, *condor_dagman* looks for *condor_submit* in the path.

DAGMAN_STORK_SUBMIT_EXE The executable that *condor_dagman* will use to submit Stork jobs. If not defined, *condor_dagman* looks for *stork_submit* in the path.

DAGMAN_CONDOR_RM_EXE The executable that *condor_dagman* will use to remove Condor jobs. If not defined, *condor_dagman* looks for *condor_rm* in the path.

DAGMAN_STORK_RM_EXE The executable that *condor_dagman* will use to remove Stork jobs. If not defined, *condor_dagman* looks for *stork_rm* in the path.

DAGMAN_PROHIBIT_MULTI_JOBS A boolean value that controls whether *condor_dagman* prohibits node job submit description files that queue multiple job procs other than parallel universe. If a DAG references such a submit file, the DAG will abort during the initialization process. If not defined, DAGMAN_PROHIBIT_MULTI_JOBS defaults to False.

DAGMAN_LOG_ON_NFS_IS_ERROR A boolean value that controls whether *condor_dagman* prohibits node job submit description files with user log files on NFS. If a DAG references such a submit description file and DAGMAN_LOG_ON_NFS_IS_ERROR is True, the DAG will abort during the initialization process. If DAGMAN_LOG_ON_NFS_IS_ERROR is False, a warning will be issued, but the DAG will still be submitted. It is *strongly* recommended that DAGMAN_LOG_ON_NFS_IS_ERROR remain set to the default value, because running a DAG with node job log files on NFS will often cause errors. If not defined, DAGMAN_LOG_ON_NFS_IS_ERROR defaults to True.

DAGMAN_ABORT_DUPLICATES A boolean value that controls whether to attempt to abort duplicate instances of *condor_dagman* running the same DAG on the same machine. When *condor_dagman* starts up, if no DAG lock file exists, *condor_dagman* creates the lock file and writes its PID into it. If the lock file does exist, and DAGMAN_ABORT_DUPLICATES is set to True, *condor_dagman* checks whether a process with the given PID exists, and if so, it assumes that there is already another instance of *condor_dagman* running the same DAG. Note that this test is not foolproof: it is possible that, if *condor_dagman* crashes, the same PID gets reused by another process before *condor_dagman* gets rerun on that DAG. This should be quite rare, however. If not defined, DAGMAN_ABORT_DUPLICATES defaults to True.

DAGMAN_SUBMIT_DEPTH_FIRST A boolean value that controls whether to submit ready DAG node jobs in (more-or-less) depth first order, as opposed to breadth-first order. Setting DAGMAN_SUBMIT_DEPTH_FIRST to True does *not* override dependencies defined in the DAG. Rather, it causes newly ready nodes to be added to the head, rather than the tail, of the ready node list. If there are no PRE scripts in the DAG, this will cause the ready nodes to be submitted depth-first. If there are PRE scripts, the order will not be strictly depth-first, but it will tend to favor depth rather than breadth in executing the DAG. If DAGMAN_SUBMIT_DEPTH_FIRST is set to True, consider also setting DAGMAN_RETRY_SUBMIT_FIRST and DAGMAN_RETRY_NODE_FIRST to True. If not defined, DAGMAN_SUBMIT_DEPTH_FIRST defaults to False.

DAGMAN_ON_EXIT_REMOVE Defines the OnExitRemove ClassAd expression placed into the *condor_dagman* submit description file by *condor_submit_dag*. The default expression is designed to ensure that *condor_dagman* is automatically re-queued by the *condor_schedd* daemon if it exits abnormally or is killed (for example, during a reboot). If this results in *condor_dagman* staying in the queue when it should exit, consider changing to a less restrictive expression, as in the example

```
(ExitBySignal == false || ExitSignal =!= 9)
```

If not defined, DAGMAN_ON_EXIT_REMOVE defaults to the expression

```
( ExitSignal =?= 11 || (ExitCode =!= UNDEFINED && ExitCode >=0 && ExitCode <= 2))
```

DAGMAN_ABORT_ON_SCARY_SUBMIT A boolean value that controls whether to abort a DAG upon detection of a scary submit event. An example of a scary submit event is one in which

the Condor ID does not match the expected value. Note that in all Condor versions prior to 6.9.3, *condor_dagman* did *not* abort a DAG upon detection of a scary submit event. This behavior is what now happens if DAGMAN_ABORT_ON_SCARY_SUBMIT is set to False. If not defined, DAGMAN_ABORT_ON_SCARY_SUBMIT defaults to True.

DAGMAN_PENDING_REPORT_INTERVAL An integer value representing the number of seconds that controls how often *condor_dagman* will print a report of pending nodes to the dagman.out file. The report will only be printed if *condor_dagman* has been waiting at least DAGMAN_PENDING_REPORT_INTERVAL seconds without seeing any node job user log events, in order to avoid cluttering the dagman.out file. This feature is mainly intended to help diagnose *condor_dagman* processes that are stuck waiting indefinitely for a job to finish. If not defined, DAGMAN_PENDING_REPORT_INTERVAL defaults to 600 seconds (10 minutes).

DAGMAN_INSERT_SUB_FILE A file name of a file containing submit description file commands to be inserted into the .condor.sub file created by *condor_submit_dag*. The specified file is inserted into the .condor.sub file before the **queue** command and before any commands specified with the **-append** *condor_submit_dag* command line option. Note that the DAGMAN_INSERT_SUB_FILE value can be overridden by the *condor_submit_dag* **-insert_sub_file** command line option.

DAGMAN_OLD_RESCUE A boolean value that controls whether *condor_dagman* uses Rescue DAG naming as defined in Condor versions from before 7.1.0 when creating a Rescue DAG. When True, the older style naming is used. In this older style Rescue DAG file naming, if a DAG input file is named my.dag, the rescue DAG file will be my.dag.rescue, and that file will be overwritten if the re-run my.dag fails again. With the current Rescue DAG file naming, the first time a Rescue DAG is created for my.dag, it will be named my.dag.rescue001, and subsequent failures of my.dag will produce Rescue DAGs named my.dag.rescue002, my.dag.rescue003, etc. If not defined, DAGMAN_OLD_RESCUE defaults to False.

DAGMAN_AUTO_RESCUE A boolean value that controls whether *condor_dagman* automatically runs Rescue DAGs. If DAGMAN_AUTO_RESCUE is True and the DAG input file my.dag is submitted, and if a Rescue DAG such as the examples my.dag.rescue001 or my.dag.rescue002 exists, then the largest magnitude Rescue DAG will be run. If not defined, DAGMAN_AUTO_RESCUE defaults to True.

Note: having DAGMAN_OLD_RESCUE and DAGMAN_AUTO_RESCUE both set to True is a fatal error.

DAGMAN_MAX_RESCUE_NUM An integer value that controls the maximum rescue DAG number that will be written, in the case that DAGMAN_OLD_RESCUE is False, or run if DAGMAN_AUTO_RESCUE is True. The maximum legal value is 999; the minimum value is 0, which prevents a rescue DAG from being written at all, or automatically run. If not defined, DAGMAN_MAX_RESCUE_NUM defaults to 100.

DAGMAN_COPY_TO_SPOOL A boolean value that when True copies the *condor_dagman* binary to the spool directory when a DAG is submitted. Setting this variable to True allows long-running DAGs to survive a DAGMan version upgrade. For running large numbers of small DAGs, leave this variable unset or set it to False. The default value if not defined is False.

DAGMAN_DEFAULT_NODE_LOG The name of a file to be used as a user log by any node jobs that do not define their own log files. The default value if not defined is `<DagFile>.nodes.log`, where `<DagFile>` is replaced by the command line argument to *condor_submit_dag* that specifies the DAG input file.

DAGMAN_GENERATE_SUBDAG_SUBMITS A boolean value specifying whether *condor_dagman* itself should create the `.condor.sub` files for nested DAGs. If set to `False`, nested DAGs will fail unless the `.condor.sub` files are generated manually by running *condor_submit_dag -no_submit* on each nested DAG, or the *-do_recurse* flag is passed to *condor_submit_dag* for the top-level DAG. DAG nodes specified with the `SUBDAG EXTERNAL` keyword or with submit description file names ending in `.condor.sub` are considered nested DAGs. The default value if not defined is `True`.

DAGMAN_MAX_JOB_HOLDS An integer value defining the maximum number of times a node job is allowed to go on hold. As a job goes on hold this number of times, it is removed from the queue. For example, if the value is 2, as the job goes on hold for the second time, it will be removed. At this time, this feature is not fully compatible with node jobs that have more than one `ProcID`. The number of holds of each process in the cluster count towards the total, rather than counting individually. So, this setting should take that possibility into account, possibly using a larger value. A value of 0 allows a job to go on hold any number of times. The default value if not defined is 100.

DAGMAN_VERBOSITY An integer value defining the verbosity of output to the `dagman.out` file, as follows (each level includes all output from lower debug levels):

- level = 0; never produce output, except for usage info
- level = 1; very quiet, output severe errors
- level = 2; output errors and warnings
- level = 3; normal output
- level = 4; internal debugging output
- level = 5; internal debugging output; outer loop debugging
- level = 6; internal debugging output; inner loop debugging
- level = 7; internal debugging output; rarely used

The default value if not defined is 3.

DAGMAN_MAX_PRE_SCRIPTS An integer defining the maximum number of PRE scripts that any given *condor_dagman* will run at the same time. The default value if not defined is 0, which means to allow any number of PRE scripts to run.

DAGMAN_MAX_POST_SCRIPTS An integer defining the maximum number of POST scripts that any given *condor_dagman* will run at the same time. The default value if not defined is 0, which means to allow any number of POST scripts to run.

DAGMAN_ALLOW_LOG_ERROR A boolean value defining whether *condor_dagman* will still attempt to run a node job, even if errors are detected in the user log specification. This setting has an effect only on nodes that are Stork jobs (not Condor jobs). The default value if not defined is `False`.

3.3.27 Configuration File Entries Relating to Security

These macros affect the secure operation of Condor. Many of these macros are described in section 3.6 on Security.

SEC_*_AUTHENTICATION This section has not yet been written

SEC_*_ENCRYPTION This section has not yet been written

SEC_*_INTEGRITY This section has not yet been written

SEC_*_NEGOTIATION This section has not yet been written

SEC_*_AUTHENTICATION_METHODS This section has not yet been written

SEC_*_CRYPTO_METHODS This section has not yet been written

GSI_DAEMON_NAME This configuration variable is retired. Instead use `ALLOW_CLIENT` or `DENY_CLIENT` as appropriate. When used, this variable defined a comma separated list of the subject name(s) of the certificate(s) that the daemons use.

GSI_DAEMON_DIRECTORY A directory name used in the construction of complete paths for the configuration variables `GSI_DAEMON_CERT`, `GSI_DAEMON_KEY`, and `GSI_DAEMON_TRUSTED_CA_DIR`, for any of these configuration variables are not explicitly set.

GSI_DAEMON_CERT A complete path and file name to the X.509 certificate to be used in GSI authentication. If this configuration variable is not defined, and `GSI_DAEMON_DIRECTORY` is defined, then Condor uses `GSI_DAEMON_DIRECTORY` to construct the path and file name as

```
GSI_DAEMON_CERT  = $(GSI_DAEMON_DIRECTORY)/hostcert.pem
```

GSI_DAEMON_KEY A complete path and file name to the X.509 private key to be used in GSI authentication. If this configuration variable is not defined, and `GSI_DAEMON_DIRECTORY` is defined, then Condor uses `GSI_DAEMON_DIRECTORY` to construct the path and file name as

```
GSI_DAEMON_KEY  = $(GSI_DAEMON_DIRECTORY)/hostkey.pem
```

GSI_DAEMON_TRUSTED_CA_DIR The directory that contains the list of trusted certification authorities to be used in GSI authentication. The files in this directory are the public keys and signing policies of the trusted certification authorities. If this configuration variable is not defined, and `GSI_DAEMON_DIRECTORY` is defined, then Condor uses `GSI_DAEMON_DIRECTORY` to construct the directory path as

```
GSI_DAEMON_TRUSTED_CA_DIR  = $(GSI_DAEMON_DIRECTORY)/certificates
```

GSI_DAEMON_PROXY A complete path and file name to the X.509 proxy to be used in GSI authentication. When this configuration variable is defined, use of this proxy takes precedence over use of a certificate and key.

DELEGATE_JOB_GSI_CREDENTIALS A boolean value that defaults to `True` for Condor version 6.7.19 and more recent versions. When `True`, a job's GSI X.509 credentials are delegated, instead of being copied. This results in a more secure communication when not encrypted.

DELEGATE_FULL_JOB_GSI_CREDENTIALS A boolean value that controls whether Condor will delegate a full or limited GSI X.509 proxy. The default value of `False` indicates the limited GSI X.509 proxy.

DELEGATE_JOB_GSI_CREDENTIALS_LIFETIME An integer value that specifies the maximum number of seconds for which delegated proxies should be valid. The default value is one day. A value of 0 indicates that the delegated proxy should be valid for as long as allowed by the credential used to create the proxy. The job may override this configuration setting by using the **delegated_job_GSI_credentials_lifetime** submit file command. This configuration variable currently only applies to proxies delegated for non-grid jobs and Condor-C jobs. It does not currently apply to globus grid jobs, which always behave as though the value is 0. This variable has no effect if `DELEGATE_JOB_GSI_CREDENTIALS` is `False`.

DELEGATE_JOB_GSI_CREDENTIALS_REFRESH A floating point number between 0 and 1 that indicates the fraction of a proxy's lifetime at which point delegated credentials with a limited lifetime should be renewed. The renewal is attempted periodically at or near the specified fraction of the lifetime of the delegated credential. The default value is 0.25. This setting has no effect if `DELEGATE_JOB_GSI_CREDENTIALS` is `False` or if `DELEGATE_JOB_GSI_CREDENTIALS_LIFETIME` is 0. For non-grid jobs, the precise timing of the proxy refresh depends on `SHADOW_CHECKPROXY_INTERVAL` . To ensure that the delegated proxy remains valid, the interval for checking the proxy should be, at most, half of the interval for refreshing it.

GRIDMAP The complete path and file name of the Globus Gridmap file. The Gridmap file is used to map X.509 distinguished names to Condor user ids.

SEC_<access-level>_SESSION_DURATION The amount of time in seconds before a communication session expires. A session is a record of necessary information to do communication between a client and daemon, and is protected by a shared secret key. The session expires to reduce the window of opportunity where the key may be compromised by attack. A short session duration increases the frequency with which daemons have to reauthenticate with each other, which may impact performance.

If the client and server are configured with different durations, the shorter of the two will be used. The default for daemons is 86400 seconds (1 day) and the default for command-line tools is 60 seconds. The shorter default for command-line tools is intended to prevent daemons

from accumulating a large number of communication sessions from the short-lived tools that contact them over time. A large number of security sessions consumes a large amount of memory. It is therefore important when changing this configuration setting to preserve the small session duration for command-line tools.

One example of how to safely change the session duration is to explicitly set a short duration for tools and *condor_submit* and a longer duration for everything else:

```
SEC_DEFAULT_SESSION_DURATION = 50000
TOOL.SEC_DEFAULT_SESSION_DURATION = 60
SUBMIT.SEC_DEFAULT_SESSION_DURATION = 60
```

Another example of how to safely change the session duration is to explicitly set the session duration for a specific daemon:

```
COLLECTOR.SEC_DEFAULT_SESSION_DURATION = 50000
```

SEC_<access-level>_SESSION_LEASE The maximum number of seconds an unused security session will be kept in a daemon's session cache before being removed to save memory. The default is 3600. If the server and client have different configurations, the smaller one will be used.

SEC_INVALIDATE_SESSIONS_VIA_TCP Use TCP (if True) or UDP (if False) for responding to attempts to use an invalid security session. This happens, for example, if a daemon restarts and receives incoming commands from other daemons that are still using a previously established security session. The default is True.

FS_REMOTE_DIR The location of a file visible to both server and client in Remote File System authentication. The default when not defined is the directory `/shared/scratch/tmp`.

ENCRYPT_EXECUTE_DIRECTORY The execute directory for jobs on Windows platforms may be encrypted by setting this configuration variable to `True`. Defaults to `False`. The method of encryption uses the EFS (Encrypted File System) feature of Windows NTFS v5.

SEC_TCP_SESSION_TIMEOUT The length of time in seconds until the timeout on individual network operations when establishing a UDP security session via TCP. The default value is 20 seconds. Scalability issues with a large pool would be the only basis for a change from the default value.

SEC_TCP_SESSION_DEADLINE An integer representing the total length of time in seconds until giving up when establishing a security session. Whereas `SEC_TCP_SESSION_TIMEOUT` specifies the timeout for individual blocking operations (connect, read, write), this setting specifies the total time across all operations, including non-blocking operations that have little cost other than holding open the socket. The default value is 120 seconds. The intention of this setting is to avoid waiting for hours for a response in the rare event that the other side freezes up and the socket remains in a connected state. This problem has been observed in some types of operating system crashes.

SEC_DEFAULT_AUTHENTICATION_TIMEOUT The length of time in seconds that Condor should attempt authenticating network connections before giving up. The default is 20 seconds. Like other security settings, the portion of the configuration variable name, `DEFAULT`, may be replaced by a different access level to specify the timeout to use for different types of commands, for example `SEC_CLIENT_AUTHENTICATION_TIMEOUT`.

SEC_PASSWORD_FILE For Unix machines, the path and file name of the file containing the pool password for password authentication.

AUTH_SSL_SERVER_CAFILE The path and file name of a file containing one or more trusted CA's certificates for the server side of a communication authenticating with SSL.

AUTH_SSL_CLIENT_CAFILE The path and file name of a file containing one or more trusted CA's certificates for the client side of a communication authenticating with SSL.

AUTH_SSL_SERVER_CADIR The path to a directory that may contain the certificates (each in its own file) for multiple trusted CAs for the server side of a communication authenticating with SSL. When defined, the authenticating entity's certificate is utilized to identify the trusted CA's certificate within the directory.

AUTH_SSL_CLIENT_CADIR The path to a directory that may contain the certificates (each in its own file) for multiple trusted CAs for the client side of a communication authenticating with SSL. When defined, the authenticating entity's certificate is utilized to identify the trusted CA's certificate within the directory.

AUTH_SSL_SERVER_CERTFILE The path and file name of the file containing the public certificate for the server side of a communication authenticating with SSL.

AUTH_SSL_CLIENT_CERTFILE The path and file name of the file containing the public certificate for the client side of a communication authenticating with SSL.

AUTH_SSL_SERVER_KEYFILE The path and file name of the file containing the private key for the server side of a communication authenticating with SSL.

AUTH_SSL_CLIENT_KEYFILE The path and file name of the file containing the private key for the client side of a communication authenticating with SSL.

CERTIFICATE_MAPFILE A path and file name of the unified map file.

SEC_ENABLE_MATCH_PASSWORD_AUTHENTICATION This is a special authentication mechanism designed to minimize overhead in the *condor_schedd* when communicating with the execute machine. Essentially, matchmaking results in a secret being shared between the *condor_schedd* and *condor_startd*, and this is used to establish a strong security session between the execute and submit daemons without going through the usual security negotiation protocol. This is especially important when operating at large scale over high latency networks (e.g. a glidein pool with one schedd and thousands of startds on a network with 0.1 second round trip times).

The default value for this configuration option is `False`. To have any effect, it must be `True` in the configuration of both the execute side (startd) as well as the submit side (schedd).

When this authentication method is used, all other security negotiation between the submit and execute daemons is bypassed. All inter-daemon communication between the submit and execute side will use the startd's settings for SEC_DAEMON_ENCRYPTION and SEC_DAEMON_INTEGRITY; the configuration of these values in the schedd, shadow, and starter are ignored.

Important: For strong security, at least one of the two, integrity or encryption, should be enabled in the startd configuration. Also, some form of strong mutual authentication (e.g. GSI) should be enabled between all daemons and the central manager or the shared secret which is exchanged in matchmaking cannot be safely encrypted when transmitted over the network.

The schedd and shadow will be authenticated as `submit-side@matchsession` when they talk to the startd and starter. The startd and starter will be authenticated as `execute-side@matchsession` when they talk to the schedd and shadow. On the submit side, authorization of the execute side happens automatically. On the execute side, it is necessary to explicitly authorize the submit side. Example:

```
ALLOW_DAEMON = submit-side@matchsession/192.168.123.*
```

Replace the example netmask with something suitable for your situation.

KERBEROS_SERVER_KEYTAB The path and file name of the keytab file that holds the necessary Kerberos principals. If not defined, this variable's value is set by the installed Kerberos; it is `/etc/v5srvtab` on most systems.

KERBEROS_SERVER_PRINCIPAL An exact Kerberos principal to use. The default value is `host/<hostname>@<realm>`, as set by the installed Kerberos. Where both KERBEROS_SERVER_PRINCIPAL and KERBEROS_SERVER_SERVICE are defined, this value takes precedence.

KERBEROS_SERVER_USER The user name that the Kerberos server principal will map to after authentication. The default value is `condor`.

KERBEROS_SERVER_SERVICE A string representing the Kerberos service name. This string is prepended with a slash character (/) and the host name in order to form the Kerberos server principal. This value defaults to `host`, resulting in the same default value as specified by using KERBEROS_SERVER_PRINCIPAL. Where both KERBEROS_SERVER_PRINCIPAL and KERBEROS_SERVER_SERVICE are defined, the value of KERBEROS_SERVER_PRINCIPAL takes precedence.

KERBEROS_CLIENT_KEYTAB The path and file name of the keytab file for the client in Kerberos authentication. This variable has no default value.

3.3.28 Configuration File Entries Relating to PrivSep

PRIVSEP_ENABLED A boolean variable that, when `True`, enables PrivSep. When `True`, the *condor_procd* is used, ignoring the definition of the configuration variable USE_PROCD . The default value when this configuration variable is not defined is `False`.

PRIVSEP_SWITCHBOARD The full (trusted) path and file name of the *condor_root_switchboard* executable.

3.3.29 Configuration File Entries Relating to Virtual Machines

These macros affect how Condor runs **vm** universe jobs on a matched machine within the pool. They specify items related to the *condor_vm-gahp*.

VM_GAHP_SERVER The complete path and file name of the *condor_vm-gahp*. There is no default value for this required configuration variable.

VM_GAHP_LOG The complete path and file name of the *condor_vm-gahp* log. If not specified on a Unix platform, the *condor_starter* log will be used for *condor_vm-gahp* log items. There is no default value for this required configuration variable on Windows platforms.

MAX_VM_GAHP_LOG Controls the maximum length (in bytes) to which the *condor_vm-gahp* log will be allowed to grow.

VM_TYPE Specifies the type of supported virtual machine software. It will be the value `kvm`, `xen` or `vmware`. There is no default value for this required configuration variable.

VM_MEMORY An integer to specify the maximum amount of memory in Mbytes that will be allowed to the virtual machine program.

VM_MAX_NUMBER An integer limit on the number of executing virtual machines. When not defined, the default value is the same `NUM_CPUS`. When it evaluates to `Undefined`, as is the case when not defined with a numeric value, no meaningful limit is imposed.

VM_STATUS_INTERVAL An integer number of seconds that defaults to 60, representing the interval between job status checks by the *condor_starter* to see if the job has finished. A minimum value of 30 seconds is enforced.

VM_GAHP_REQ_TIMEOUT An integer number of seconds that defaults to 300 (five minutes), representing the amount of time Condor will wait for a command issued from the *condor_starter* to the *condor_vm-gahp* to be completed. When a command times out, an error is reported to the *condor_startd*.

VM_RECHECK_INTERVAL An integer number of seconds that defaults to 600 (ten minutes), representing the amount of time the *condor_startd* waits after a virtual machine error as reported by the *condor_starter*, and before checking a final time on the status of the virtual machine. If the check fails, Condor disables starting any new vm universe jobs by removing the `VM_Type` attribute from the machine ClassAd.

VM_SOFT_SUSPEND A boolean value that defaults to `False`, causing Condor to free the memory of a vm universe job when the job is suspended. When `True`, the memory is not freed.

VM_UNIV_NOBODY_USER Identifies a login name of a user with a home directory that may be used for job owner of a vm universe job. The `nobody` user normally utilized when the job arrives from a different UID domain will not be allowed to invoke a VMware virtual machine.

ALWAYS_VM_UNIV_USE_NOBODY A boolean value that defaults to `False`. When `True`, all vm universe jobs (independent of their UID domain) will run as the user defined in `VM_UNIV_NOBODY_USER`.

VM_NETWORKING A boolean variable describing if networking is supported. When not defined, the default value is `False`.

VM_NETWORKING_TYPE A string describing the type of networking, required and relevant only when `VM_NETWORKING` is `True`. Defined strings are

```
bridge
nat
nat, bridge
```

VM_NETWORKING_DEFAULT_TYPE Where multiple networking types are given in `VM_NETWORKING_TYPE`, this optional configuration variable identifies which to use. Therefore, for

```
VM_NETWORKING_TYPE = nat, bridge
```

this variable may be defined as either `nat` or `bridge`. Where multiple networking types are given in `VM_NETWORKING_TYPE`, and this variable is *not* defined, a default of `nat` is used.

VM_NETWORKING_BRIDGE_INTERFACE For Xen and KVM only, a required string if bridge networking is to be enabled. It specifies the networking interface that vm universe jobs will use.

The following configuration variables are specific to the VMware virtual machine software.

VMWARE_PERL The complete path and file name to *Perl*. There is no default value for this required variable.

VMWARE_SCRIPT The complete path and file name of the script that controls VMware. There is no default value for this required variable.

VMWARE_NETWORKING_TYPE An optional string used in networking that the *condor_vm-gahp* inserts into the VMware configuration file to define a networking type. Defined types are `nat` or `bridged`. If a default value is needed, the inserted string will be `nat`.

VMWARE_NAT_NETWORKING_TYPE An optional string used in networking that the *condor_vm-gahp* inserts into the VMware configuration file to define a networking type. If nat networking is used, this variable's definition takes precedence over one defined by `VMWARE_NETWORKING_TYPE`.

VMWARE_BRIDGE_NETWORKING_TYPE An optional string used in networking that the *condor_vm-gahp* inserts into the VMware configuration file to define a networking type. If bridge networking is used, this variable's definition takes precedence over one defined by VMWARE_NETWORKING_TYPE.

VMWARE_LOCAL_SETTINGS_FILE The complete path and file name to a file, whose contents will be inserted into the VMware description file (i.e., the .vmx file) before Condor starts the virtual machine. This parameter is optional.

The following configuration variables are specific to the Xen virtual machine software.

XEN_BOOTLOADER A required full path and executable for the Xen bootloader, if the kernel image includes a disk image.

XEN_LOCAL_SETTINGS_FILE A complete path and file name. The file's contents will be included in the Xen configuration file that Condor writes to run the virtual machine. This parameter is optional.

The following two macros affect the configuration of Condor where Condor is running on a host machine, the host machine is running an inner virtual machine, and Condor is also running on that inner virtual machine. These two variables have nothing to do with the **vm** universe.

VMP_HOST_MACHINE A configuration variable for the inner virtual machine, which specifies the host name.

VMP_VM_LIST For the host, a comma separated list of the host names or IP addresses for machines running inner virtual machines on a host.

3.3.30 Configuration File Entries Relating to High Availability

These macros affect the high availability operation of Condor.

MASTER_HA_LIST Similar to DAEMON_LIST, this macro defines a list of daemons that the *condor_master* starts and keeps its watchful eyes on. However, the MASTER_HA_LIST daemons are run in a *High Availability* mode. The list is a comma or space separated list of subsystem names (as listed in section 3.3.1). For example,

```
MASTER_HA_LIST = SCHEDD
```

The *High Availability* feature allows for several *condor_master* daemons (most likely on separate machines) to work together to insure that a particular service stays available. These *condor_master* daemons ensure that one and only one of them will have the listed daemons running.

To use this feature, the lock URL must be set with HA_LOCK_URL.

Currently, only file URLs are supported (those with file:...). The default value for MASTER_HA_LIST is the empty string, which disables the feature.

HA_LOCK_URL This macro specifies the URL that the *condor_master* processes use to synchronize for the *High Availability* service. Currently, only file URLs are supported; for example, file:/share/spool. Note that this URL must be identical for all *condor_master* processes sharing this resource. For *condor_schedd* sharing, we recommend setting up SPOOL on an NFS share and having all *High Availability condor_schedd* processes sharing it, and setting the HA_LOCK_URL to point at this directory as well. For example:

```
MASTER_HA_LIST = SCHEDD
SPOOL = /share/spool
HA_LOCK_URL = file:/share/spool
VALID_SPOOL_FILES = SCHEDD.lock
```

A separate lock is created for each *High Availability* daemon.

There is no default value for HA_LOCK_URL.

Lock files are in the form <SUBSYS>.lock. *condor_preen* is not currently aware of the lock files and will delete them if they are placed in the SPOOL directory, so be sure to add <SUBSYS>.lock to VALID_SPOOL_FILES for each *High Availability* daemon.

HA_<SUBSYS>_LOCK_URL This macro controls the *High Availability* lock URL for a specific subsystem as specified in the configuration variable name, and it overrides the system-wide lock URL specified by HA_LOCK_URL. If not defined for each subsystem, HA_<SUBSYS>_LOCK_URL is ignored, and the value of HA_LOCK_URL is used.

HA_LOCK_HOLD_TIME This macro specifies the number of seconds that the *condor_master* will hold the lock for each *High Availability* daemon. Upon gaining the shared lock, the *condor_master* will hold the lock for this number of seconds. Additionally, the *condor_master* will periodically renew each lock as long as the *condor_master* and the daemon are running. When the daemon dies, or the *condor_master* exists, the *condor_master* will immediately release the lock(s) it holds.

HA_LOCK_HOLD_TIME defaults to 3600 seconds (one hour).

HA_<SUBSYS>_LOCK_HOLD_TIME This macro controls the *High Availability* lock hold time for a specific subsystem as specified in the configuration variable name, and it overrides the system wide poll period specified by HA_LOCK_HOLD_TIME. If not defined for each subsystem, HA_<SUBSYS>_LOCK_HOLD_TIME is ignored, and the value of HA_LOCK_HOLD_TIME is used.

HA_POLL_PERIOD This macro specifies how often the *condor_master* polls the *High Availability* locks to see if any locks are either stale (meaning not updated for HA_LOCK_HOLD_TIME seconds), or have been released by the owning *condor_master*. Additionally, the *condor_master* renews any locks that it holds during these polls.

HA_POLL_PERIOD defaults to 300 seconds (five minutes).

HA_<SUBSYS>_POLL_PERIOD This macro controls the *High Availability* poll period for a specific subsystem as specified in the configuration variable name, and it overrides the system wide poll period specified by HA_POLL_PERIOD. If not defined for each subsystem, HA_<SUBSYS>_POLL_PERIOD is ignored, and the value of HA_POLL_PERIOD is used.

MASTER_<SUBSYS>_CONTROLLER Used only in HA configurations involving the *condor_had*.

The *condor_master* has the concept of a controlling and controlled daemon, typically with the *condor_had* daemon serving as the controlling process. In this case, all *condor_on* and *condor_off* commands directed at controlled daemons are given to the controlling daemon, which then handles the command, and, when required, sends appropriate commands to the *condor_master* to do the actual work. This allows the controlling daemon to know the state of the controlled daemon.

As of 6.7.14, this configuration variable must be specified for all configurations using *condor_had*. To configure the *condor_negotiator* controlled by *condor_had*:

```
MASTER_NEGOTIATOR_CONTROLLER = HAD
```

The macro is named by substituting <SUBSYS> with the appropriate subsystem string as defined in section 3.3.1.

HAD_LIST A comma-separated list of all *condor_had* daemons in the form IP:port or hostname:port. Each central manager machine that runs the *condor_had* daemon should appear in this list. If HAD_USE_PRIMARY is set to True, then the first machine in this list is the primary central manager, and all others in the list are backups.

All central manager machines must be configured with an identical HAD_LIST. The machine addresses are identical to the addresses defined in COLLECTOR_HOST.

HAD_USE_PRIMARY Boolean value to determine if the first machine in the HAD_LIST configuration variable is a primary central manager. Defaults to False.

HAD_CONTROLLEE This macro is used to specify the name of the daemon which the *condor_had* daemon controls. This name should match the daemon name in the *condor_master*'s DAEMON_LIST. The default value of HAD_CONTROLLEE is "NEGOTIATOR".

HAD_CONNECTION_TIMEOUT The time (in seconds) that the *condor_had* daemon waits before giving up on the establishment of a TCP connection. The failure of the communication connection is the detection mechanism for the failure of a central manager machine. For a LAN, a recommended value is 2 seconds. The use of authentication (by Condor) increases the connection time. The default value is 5 seconds. If this value is set too low, *condor_had* daemons will incorrectly assume the failure of other machines.

HAD_ARGS Command line arguments passed by the *condor_master* daemon as it invokes the *condor_had* daemon. To make high availability work, the *condor_had* daemon requires the port number it is to use. This argument is of the form

```
-p $(HAD_PORT_NUMBER)
```

where HAD_PORT_NUMBER is a helper configuration variable defined with the desired port number. Note that this port number must be the same value here as used in HAD_LIST. There is no default value.

HAD The path to the *condor_had* executable. Normally it is defined relative to $(SBIN). This configuration variable has no default value.

MAX_HAD_LOG Controls the maximum length in bytes to which the *condor_had* daemon log will be allowed to grow. It will grow to the specified length, then be saved to a file with the suffix .old. The .old file is overwritten each time the log is saved, thus the maximum space devoted to logging is twice the maximum length of this log file. A value of 0 specifies that this file may grow without bounds. The default is 1 Mbyte.

HAD_DEBUG Logging level for the *condor_had* daemon. See <SUBSYS>_DEBUG for values.

HAD_LOG Full path and file name of the log file. There is no default value.

REPLICATION_LIST A comma-separated list of all *condor_replication* daemons in the form IP:port or hostname:port. Each central manager machine that runs the *condor_had* daemon should appear in this list. All potential central manager machines must be configured with an identical REPLICATION_LIST.

STATE_FILE A full path and file name of the file protected by the replication mechanism. When not defined, the default path and file used is

```
$(SPOOL)/Accountantnew.log
```

REPLICATION_INTERVAL Sets how often the *condor_replication* daemon initiates its tasks of replicating the $(STATE_FILE). It is defined in seconds and defaults to 300 (5 minutes). This is the same as the default NEGOTIATOR_INTERVAL.

MAX_TRANSFERER_LIFETIME A timeout period within which the process that transfers the state file must complete its transfer. The recommended value is 2 * average size of state file / network rate. It is defined in seconds and defaults to 300 (5 minutes).

HAD_UPDATE_INTERVAL Like UPDATE_INTERVAL, determines how often the *condor_had* is to send a ClassAd update to the *condor_collector*. Updates are also sent at each and every change in state. It is defined in seconds and defaults to 300 (5 minutes).

HAD_USE_REPLICATION A boolean value that defaults to False. When True, the use of *condor_replication* daemons is enabled.

REPLICATION_ARGS Command line arguments passed by the *condor_master* daemon as it invokes the *condor_replication* daemon. To make high availability work, the *condor_replication* daemon requires the port number it is to use. This argument is of the form

```
-p $(REPLICATION_PORT_NUMBER)
```

where REPLICATION_PORT_NUMBER is a helper configuration variable defined with the desired port number. Note that this port number must be the same value as used in REPLICATION_LIST. There is no default value.

REPLICATION The full path and file name of the *condor_replication* executable. It is normally defined relative to $(SBIN). There is no default value.

MAX_REPLICATION_LOG Controls the maximum length in bytes to which the *condor_replication* daemon log will be allowed to grow. It will grow to the specified length, then be saved to a file with the suffix .old. The .old file is overwritten each time the log is saved, thus the maximum space devoted to logging is twice the maximum length of this log file. A value of 0 specifies that this file may grow without bounds. The default is 1 Mbyte.

REPLICATION_DEBUG Logging level for the *condor_replication* daemon. See <SUBSYS>_DEBUG for values.

REPLICATION_LOG Full path and file name to the log file. There is no default value.

TRANSFERER The full path and file name of the *condor_transferer* executable. Versions of Condor previous to 7.2.2 hard coded the location as $(RELEASE_DIR)/sbin/condor_transferer. This is now the default value. The future default value is likely to change, and be defined relative to $(SBIN).

TRANSFERER_LOG Full path and file name to the log file. There is no default value for this variable; a definition is required if the *condor_replication* daemon does a file transfer.

TRANSFERER_DEBUG Logging level for the *condor_transferer* daemon. See <SUBSYS>_DEBUG for values.

MAX_TRANSFERER_LOG Controls the maximum length in bytes to which the *condor_transferer* daemon log will be allowed to grow. A value of 0 specifies that this file may grow without bounds. The default is 1 Mbyte.

3.3.31 Configuration File Entries Relating to Quill

These macros affect the Quill database management and interface to its representation of the job queue.

QUILL The full path name to the *condor_quill* daemon.

QUILL_ARGS Arguments to be passed to the *condor_quill* daemon upon its invocation.

QUILL_LOG Path to the Quill daemon's log file.

QUILL_ENABLED A boolean variable that defaults to False. When True, Quill functionality is enabled. When False, the Quill daemon writes a message to its log and exits. The *condor_q* and *condor_history* tools then do not use Quill.

QUILL_NAME A string that uniquely identifies an instance of the *condor_quill* daemon, as there may be more than *condor_quill* daemon per pool. The string must not be the same as for any *condor_schedd* daemon.

See the description of MASTER_NAME in section 3.3.9 on page 194 for defaults and composition of valid Condor daemon names.

QUILL_USE_SQL_LOG In order for Quill to store historical job information or resource information, the Condor daemons must write information to the SQL logfile. By default, this is set to False, and the only information Quill stores in the database is the current job queue. This can be set on a per daemon basis. For example, to store information about historical jobs, but not store execute resource information, set QUILL_USE_SQL_LOG to False and set SCHEDD._QUILL_USE_SQL_LOG to True.

QUILL_DB_NAME A string that identifies a database within a database server.

QUILL_DB_USER A string that identifies the *PostgreSQL* user that Quill will connect to the database as. We recommend "**quillwriter**" for this setting.

QUILL_DB_TYPE A string that distinguishes between database system types. Defaults to the only database system currently defined, "PGSQL".

QUILL_DB_IP_ADDR The host address of the database server. It can be either an IP address or an IP address. It must match exactly what is used in the .pgpass file.

QUILL_POLLING_PERIOD The frequency, in number of seconds, at which the Quill daemon polls the file job_queue.log for updates. New information in the log file is sent to the database. The default value is 10.

QUILL_NOT_RESPONDING_TIMEOUT The length of time, in seconds, before the *condor_master* may decide that the *condor_quill* daemon is hung due to a lack of communication, potentially causing the *condor_master* to kill and restart the *condor_quill* daemon. When the *condor_quill* daemon is processing a very long log file, it may not be able to communicate with the master. The default is 3600 seconds, or one hour. It may be advisable to increase this to several hours.

QUILL_MAINTAIN_DB_CONN A boolean variable that defaults to True. When True, the *condor_quill* daemon maintains an open connection the database server, which speeds up updates to the database. As each open connection consumes resources at the database server, we recommend a setting of False for large pools.

DATABASE_PURGE_INTERVAL The interval, in seconds, between scans of the database to identify and delete records that are beyond their history durations. The default value is 86400, or one day.

DATABASE_REINDEX_INTERVAL The interval, in seconds, between reindex commands on the database. The default value is 86400, or one day. This is only used when the QUILL_DB_TYPE is set to "PGSQL".

QUILL_JOB_HISTORY_DURATION The number of days after entry into the database that a job will remain in the database. After QUILL_JOB_HISTORY_DURATION days, the job is deleted. The job history is the final ClassAd, and contains all information necessary for *condor_history* to succeed. The default is 3650, or about 10 years.

QUILL_RUN_HISTORY_DURATION The number of days after entry into the database that extra information about the job will remain in the database. After QUILL_RUN_HISTORY_DURATION days, the records are deleted. This data includes matches made for the job, file transfers the job performed, and user log events. The default is 7 days, or one week.

QUILL_RESOURCE_HISTORY_DURATION The number of days after entry into the database that a resource record will remain in the database. After QUILL_RESOURCE_HISTORY_DURATION days, the record is deleted. The resource history data includes the ClassAd of a compute slot, submitter ClassAds, and daemon ClassAds. The default is 7 days, or one week.

QUILL_DBSIZE_LIMIT After each purge, the *condor_quill* daemon estimates the size of the database. If the size of the database exceeds this limit, the *condor_quill* daemon will e-mail the administrator a warning. This size is given in gigabytes, and defaults to 20.

QUILL_MANAGE_VACUUM A boolean value that defaults to False. When True, the *condor_quill* daemon takes on the maintenance task of vacuuming the database. As of *PostgreSQL* version 8.1, the database can perform this task automatically; therefore, having the *condor_quill* daemon vacuum is not necessary. A value of True causes warnings to be written to the log file.

QUILL_SHOULD_REINDEX A boolean value that defaults to True. When True, the *condor_quill* daemon will re-index the database tables when the history file is purged of old data. So, if Quill is configured to never delete history data, the tables are never re-indexed.

QUILL_IS_REMOTELY_QUERYABLE A boolean value that defaults to True. When False, the remote database tables may not be remotely queryable.

QUILL_DB_QUERY_PASSWORD Defines the password string needed by *condor_q* to gain read access for remotely querying the Quill database.

QUILL_ADDRESS_FILE When defined, it specifies the path and file name of a local file containing the IP address and port number of the Quill daemon. By using the file, tools executed on the local machine do not need to query the central manager in order to find the *condor_quill* daemon.

DBMSD The full path name to the *condor_dbmsd* daemon. The default location is $(SBIN)/condor_dbmsd.

DBMSD_ARGS Arguments to be passed to the *condor_dbmsd* daemon upon its invocation. The default arguments are -f.

DBMSD_LOG Path to the *condor_dbmsd* daemon's log file. The default log location is $(LOG)/DbmsdLog.

DBMSD_NOT_RESPONDING_TIMEOUT The length of time, in seconds, before the *condor_master* may decide that the *condor_dbmsd* is hung due to a lack of communication, potentially causing the *condor_master* to kill and restart the *condor_dbmsd* daemon. When the *condor_dbmsd* is purging or reindexing a very large database, it may not be able to communicate with the master. The default is 3600 seconds, or one hour. It may be advisable to increase this to several hours.

3.3.32 MyProxy Configuration File Macros

In some cases, Condor can autonomously refresh GSI certificate proxies via *MyProxy*, available from http://myproxy.ncsa.uiuc.edu/.

MYPROXY_GET_DELEGATION The full path name to the *myproxy-get-delegation* executable, installed as part of the *MyProxy* software. Often, it is necessary to wrap the actual executable with a script that sets the environment, such as the `LD_LIBRARY_PATH`, correctly. If this macro is defined, Condor-G and *condor_credd* will have the capability to autonomously refresh proxy certificates. By default, this macro is undefined.

3.3.33 Configuration File Macros Affecting APIs

ENABLE_SOAP A boolean value that defaults to `False`. When `True`, Condor daemons will respond to HTTP PUT commands as if they were SOAP calls. When `False`, all HTTP PUT commands are denied.

ENABLE_WEB_SERVER A boolean value that defaults to `False`. When `True`, Condor daemons will respond to HTTP GET commands, and send the static files sitting in the subdirectory defined by the configuration variable `WEB_ROOT_DIR`. In addition, web commands are considered a READ command, so the client will be checked by host-based security.

SOAP_LEAVE_IN_QUEUE A boolean expression that when `True`, causes a job in the completed state to remain in the queue, instead of being removed based on the completion of file transfer. If provided, this expression will be logically ANDed with the default behavior of leaving the job in the queue until `FilesRetrieved` becomes `True`.

WEB_ROOT_DIR A complete path to the directory containing all the files served by the web server.

<SUBSYS>_ENABLE_SOAP_SSL A boolean value that defaults to `False`. When `True`, enables SOAP over SSL for the specified `<SUBSYS>`. Any specific `<SUBSYS>_ENABLE_SOAP_SSL` setting overrides the value of `ENABLE_SOAP_SSL`.

ENABLE_SOAP_SSL A boolean value that defaults to `False`. When `True`, enables SOAP over SSL for all daemons.

<SUBSYS>_SOAP_SSL_PORT The port number on which SOAP over SSL messages are accepted, when SOAP over SSL is enabled. The <SUBSYS> must be specified, because multiple daemons running on a single machine may not share a port. This parameter is required when SOAP over SSL is enabled. There is no default value.

The macro is named by substituting <SUBSYS> with the appropriate subsystem string as defined in section 3.3.1.

SOAP_SSL_SERVER_KEYFILE The complete path and file name to specify the daemon's identity, as used in authentication when SOAP over SSL is enabled. The file is to be an OpenSSL PEM file containing a certificate and private key. This parameter is required when SOAP over SSL is enabled. There is no default value.

SOAP_SSL_SERVER_KEYFILE_PASSWORD An optional complete path and file name to specify a password for unlocking the daemon's private key. There is no default value.

SOAP_SSL_CA_FILE The complete path and file name to specify a file containing certificates of trusted Certificate Authorities (CAs). Only clients who present a certificate signed by a trusted CA will be authenticated. When SOAP over SSL is enabled, this parameter or SOAP_SSL_CA_DIR must be set. There is no default value.

SOAP_SSL_CA_DIR The complete path to a directory containing certificates of trusted Certificate Authorities (CAs). Only clients who present a certificate signed by a trusted CA will be authenticated. When SOAP over SSL is enabled, this variable or the variable SOAP_SSL_CA_FILE must be defined. This variable is also used when communicating with an Amazon EC2 server (possibly through a proxy), although it is not required in that case. There is no default value.

SOAP_SSL_DH_FILE An optional complete path and file name to a DH file containing keys for a DH key exchange. There is no default value.

SOAP_SSL_SKIP_HOST_CHECK When a SOAP server is authenticated via SSL, the server's host name is normally compared with the host name contained in the server's X.509 credential. If the two do not match, authentication fails. When this boolean variable is set to True, the host name comparison is disabled. The default value is False.

3.3.34 Configuration File Entries Relating to *condor_ssh_to_job*

These macros affect how Condor deals with *condor_ssh_to_job*, a tool that allows users to interactively debug jobs. With these configuration variables, the administrator can control who can use the tool, and how the *ssh* programs are invoked. The manual page for *condor_ssh_to_job* is at section 9.

ENABLE_SSH_TO_JOB A boolean expression read by the *condor_starter*, that when True allows the owner of the job or a queue super user on the *condor_schedd* where the job was submitted to connect to the job via *ssh*. The expression may refer to attributes of both the job and the machine ClassAds. The job ClassAd attributes may be referenced by using the prefix TARGET., and the machine ClassAd attributes may be referenced by using the prefix MY..

When `False`, it prevents *condor_ssh_to_job* from starting an *ssh* session. The default value is `True`.

SCHEDD_ENABLE_SSH_TO_JOB A boolean expression read by the *condor_schedd*, that when `True` allows the owner of the job or a queue super user to connect to the job via *ssh* if the execute machine also allows *condor_ssh_to_job* access (see `ENABLE_SSH_TO_JOB`). The expression may refer to attributes of only the job ClassAd. When `False`, it prevents *condor_ssh_to_job* from starting an *ssh* session for all jobs managed by the *condor_schedd*. The default value is `True`.

SSH_TO_JOB_<SSH-CLIENT>_CMD A string read by the *condor_ssh_to_job* tool. It specifies the command and arguments to use when invoking the program specified by `<SSH-CLIENT>`. Values substituted for the placeholder `<SSH-CLIENT>` may be `SSH`, `SFTP`, `SCP`, or any other *ssh* client capable of using a command as a proxy for the connection to *sshd*. The entire command plus arguments string is enclosed in double quote marks. Individual arguments may be quoted with single quotes, using the same syntax as for arguments in a *condor_submit* file. The following substitutions are made within the arguments:

`%h`: is substituted by the remote host

`%i`: is substituted by the ssh key

`%k`: is substituted by the known hosts file

`%u`: is substituted by the remote user

`%x`: is substituted by a proxy command suitable for use with the *OpenSSH* ProxyCommand option

`%%`: is substituted by the percent mark character

The default string is:

```
"ssh -oUser=%u -oIdentityFile=%i -oStrictHostKeyChecking=yes
-oUserKnownHostsFile=%k -oGlobalKnownHostsFile=%k
-oProxyCommand=%x %h"
```

When the `<SSH-CLIENT>` is *scp*, %h is omitted.

SSH_TO_JOB_SSHD The path and executable name of the *ssh* daemon. The value is read by the *condor_starter*. The default value is `/usr/sbin/sshd`.

SSH_TO_JOB_SSHD_ARGS A string, read by the *condor_starter* that specifies the command-line arguments to be passed to the *sshd* to handle an incoming ssh connection on its `stdin` or `stdout` streams in inetd mode. Enclose the entire arguments string in double quote marks. Individual arguments may be quoted with single quotes, using the same syntax as for arguments in a Condor submit description file. Within the arguments, the characters `%f` are replaced by the path to the *sshd* configuration file the characters %% are replaced by a single percent character. The default value is the string `"-i -e -f %f"`.

SSH_TO_JOB_SSHD_CONFIG_TEMPLATE A string, read by the *condor_starter* that specifies the path and file name of an *sshd* configuration template file. The template is turned into an *sshd* configuration file by replacing macros within the template that

specify such things as the paths to key files. The macro replacement is done by the script `$(LIBEXEC)/condor_ssh_to_job_sshd_setup`. The default value is `$(LIB)/condor_ssh_to_job_sshd_config_template`.

SSH_TO_JOB_SSH_KEYGEN A string, read by the *condor_starter* that specifies the path to *ssh_keygen*, the program used to create ssh keys.

SSH_TO_JOB_SSH_KEYGEN_ARGS A string, read by the *condor_starter* that specifies the command-line arguments to be passed to the *ssh_keygen* to generate an ssh key. Enclose the entire arguments string in double quotes. Individual arguments may be quoted with single quotes, using the same syntax as for arguments in a Condor submit description file. Within the arguments, the characters `%f` are replaced by the path to the key file to be generated, and the characters `%%` are replaced by a single percent character. The default value is the string `"-N '' -C '' -q -f %f -t rsa"`. If the user specifies additional arguments with the command `condor_ssh_to_job -keygen-options`, then those arguments are placed after the arguments specified by the value of `SSH_TO_JOB_SSH_KEYGEN_ARGS`.

3.3.35 *condor_rooster* Configuration File Macros

condor_rooster is an optional daemon that may be added to the *condor_master* daemon's `DAEMON_LIST`. It is responsible for waking up hibernating machines when their `UNHIBERNATE` expression becomes `True`. In the typical case, a pool runs a single instance of *condor_rooster* on the central manager. However, if the network topology requires that Wake On LAN packets be sent to specific machines from different locations, *condor_rooster* can be run on any machine(s) that can read from the pool's *condor_collector* daemon.

For *condor_rooster* to wake up hibernating machines, the collecting of offline machine ClassAds must be enabled. See variable `OFFLINE_LOG` on page 206 for details on how to do this.

ROOSTER_INTERVAL The integer number of seconds between checks for offline machines that should be woken. The default value is 300.

ROOSTER_MAX_UNHIBERNATE An integer specifying the maximum number of machines to wake up per cycle. The default value of 0 means no limit.

ROOSTER_UNHIBERNATE A boolean expression that specifies which machines should be woken up. The default expression is `Offline && Unhibernate`. If network topology or other considerations demand that some machines in a pool be woken up by one instance of *condor_rooster*, while others be woken up by a different instance, `ROOSTER_UNHIBERNATE` may be set locally such that it is different for the two instances of *condor_rooster*. In this way, the different instances will only try to wake up their respective subset of the pool.

ROOSTER_UNHIBERNATE_RANK A ClassAd expression specifying which machines should be woken up first in a given cycle. Higher ranked machines are woken first. If the number of machines to be woken up is limited by `ROOSTER_MAX_UNHIBERNATE` , the rank may be used for determining which machines are woken before reaching the limit.

ROOSTER_WAKEUP_CMD A string representing the command line invoked by *condor_rooster* that is to wake up a machine. The command and any arguments should be enclosed in double quote marks, the same as **arguments** syntax in a Condor submit description file. The default value is `"$(BIN)/condor_power -d -i"`. The command is expected to read from its standard input a ClassAd representing the offline machine.

3.3.36 *condor_shared_port* Configuration File Macros

These configuration variables affect the *condor_shared_port* daemon. For general discussion of *condor_shared_port*, see 362.

SHARED_PORT_DAEMON_AD_FILE This specifies the full path and name of a file used to publish the address of *condor_shared_port*. This file is read by the other daemons that have `USE_SHARED_PORT=True` and which are therefore sharing the same port. The default typically does not need to be changed.

SHARED_PORT_MAX_WORKERS An integer that specifies the maximum number of sub-processes created by *condor_shared_port* while servicing requests to connect to the daemons that are sharing the port. The default is 50.

DAEMON_SOCKET_DIR This specifies the directory where Unix versions of Condor daemons will create named sockets so that incoming connections can be forwarded to them by *condor_shared_port*. If this directory does not exist, it will be created. The maximum length of named socket paths plus names is restricted by the operating system, so it is important that this path not exceed 90 characters.

Write access to this directory grants permission to receive connections through the shared port. By default, the directory is created to be owned by Condor and is made to be only writable by Condor. One possible reason to broaden access to this directory is if execute nodes are accessed via CCB and the submit node is behind a firewall with only one open port (the port assigned to *condor_shared_port*). In this case, commands that interact with the execute node such as *condor_ssh_to_job* will not be able to operate unless run by a user with write access to `DAEMON_SOCKET_DIR`. In this case, one could grant tmp-like permissions to this directory so that all users can receive CCB connections back through the firewall. (But consider the wisdom of having a firewall in the first place if you are going to circumvent it in this way.) The default `DAEMON_SOCKET_DIR` is `$(LOCK)/daemon_sock`. This directory must be on a local file system that supports named sockets.

SHARED_PORT_ARGS Like all daemons started by *condor_master*, *condor_shared_port* arguments can be customized. One reason to do this is to specify the port number that *condor_shared_port* should use. For example, the following line configures *condor_shared_port* to use port 4080.

```
SHARED_PORT_ARGS = -p 4080
```

If no port is specified, a port will be dynamically chosen; it may be different each time Condor is started.

3.3.37 Configuration File Entries Relating to Hooks

These macros control the various hooks that interact with Condor. Currently, there are two independent sets of hooks. One is a set of fetch work hooks, some of which are invoked by the *condor_startd* to optionally fetch work, and some are invoked by the *condor_starter*. See section 4.4.1 on page 509 on Job Hooks for more details. The other set replace functionality of the *condor_job_router* daemon. Documentation for the *condor_job_router* daemon is in section 5.6 on page 585.

SLOT<N>_JOB_HOOK_KEYWORD For the fetch work hooks, the keyword used to define which set of hooks a particular compute slot should invoke. The value of `<N>` is replaced by the slot identification number. For example, on slot 1, the variable name will be called `[SLOT1_JOB_HOOK_KEYWORD`. There is no default keyword. Sites that wish to use these job hooks must explicitly define the keyword and the corresponding hook paths.

STARTD_JOB_HOOK_KEYWORD For the fetch work hooks, the keyword used to define which set of hooks a particular *condor_startd* should invoke. This setting is only used if a slot-specific keyword is not defined for a given compute slot. There is no default keyword. Sites that wish to use job hooks must explicitly define the keyword and the corresponding hook paths.

<Keyword>_HOOK_FETCH_WORK For the fetch work hooks, the full path to the program to invoke whenever the *condor_startd* wants to fetch work. `<Keyword>` is is the hook keyword defined to distinguish between sets of hooks. There is no default.

<Keyword>_HOOK_REPLY_FETCH For the fetch work hooks, the full path to the program to invoke when the hook defined by `<Keyword>_HOOK_FETCH_WORK` returns data and the the *condor_startd* decides if it is going to accept the fetched job or not. `<Keyword>` is is the hook keyword defined to distinguish between sets of hooks.

<Keyword>_HOOK_REPLY_CLAIM For the fetch work hooks, the full path to the program to invoke whenever the *condor_startd* finishes fetching a job and decides what to do with it. `<Keyword>` is is the hook keyword defined to distinguish

between sets of hooks. There is no default.

<Keyword>_HOOK_PREPARE_JOB For the fetch work hooks, the full path to the program invoked by the *condor_starter* before it runs the job. `<Keyword>` is is the hook keyword defined to distinguish between sets of hooks.

<Keyword>_HOOK_UPDATE_JOB_INFO This configuration variable is used by both fetch work hooks and by *condor_job_router* hooks.

For the fetch work hooks, the full path to the program invoked by the *condor_starter* periodically as the job runs, allowing the *condor_starter* to present an updated and augmented job ClassAd to the program. See section 4.4.1 on page 510 for the list of additional attributes included. When the job is first invoked, the *condor_starter* will invoke the program after `$(STARTER_INITIAL_UPDATE_INTERVAL)` seconds. Thereafter, the *condor_starter* will invoke the program every `$(STARTER_UPDATE_INTERVAL)` seconds. `<Keyword>` is the hook keyword defined to distinguish between sets of hooks.

As a Job Router hook, the full path to the program invoked when the Job Router polls the status of routed jobs at intervals set by JOB_ROUTER_POLLING_PERIOD. <Keyword> is is the hook keyword defined by JOB_ROUTER_HOOK_KEYWORD to identify the hooks.

<Keyword>_HOOK_EVICT_CLAIM For the fetch work hooks, the full path to the program to invoke whenever the *condor_startd* needs to evict a fetched claim. <Keyword> is is the hook keyword defined to distinguish between sets of hooks. There is no default.

<Keyword>_HOOK_JOB_EXIT For the fetch work hooks, the full path to the program invoked by the *condor_starter* whenever a job exits, either on its own or when being evicted from an execution slot. <Keyword> is is the hook keyword defined to distinguish between sets of hooks.

FetchWorkDelay An expression that defines the number of seconds that the *condor_startd* should wait after an invocation of <Keyword>_HOOK_FETCH_WORK completes before the hook should be invoked again. The expression is evaluated in the context of the slot ClassAd, and the ClassAd of the currently running job (if any). The expression must evaluate to an integer. If not defined, the *condor_startd* will wait 300 seconds (five minutes) between attempts to fetch work. For more information about this expression, see section 4.4.1 on page 514.

JOB_ROUTER_HOOK_KEYWORD For the Job Router hooks, the keyword used to define the set of hooks the *condor_job_router* is to invoke to replace functionality of routing translation. There is no default keyword. Use of these hooks requires the explicit definition of the keyword and the corresponding hook paths.

<Keyword>_HOOK_TRANSLATE_JOB A Job Router hook, the full path to the program invoked when the Job Router has determined that a job meets the definition for a route. This hook is responsible for doing the transformation of the job. <Keyword> is is the hook keyword defined by JOB_ROUTER_HOOK_KEYWORD to identify the hooks.

<Keyword>_HOOK_JOB_FINALIZE A Job Router hook, the full path to the program invoked when the Job Router has determined that the job completed. <Keyword> is is the hook keyword defined by JOB_ROUTER_HOOK_KEYWORD to identify the hooks.

<Keyword>_HOOK_JOB_CLEANUP A Job Router hook, the full path to the program invoked when the Job Router finishes managing the job. <Keyword> is is the hook keyword defined by JOB_ROUTER_HOOK_KEYWORD to identify the hooks.

The following macros describe the *Daemon ClassAd Hook* capabilities of Condor. The Daemon ClassAd Hook mechanism is used to run executables (called jobs) directly from the *condor_startd* and *condor_schedd* daemons. The output from the jobs is incorporated into the machine ClassAd generated by the respective daemon. The mechanism is described in section 4.4.3 on page 518.

STARTD_CRON_NAME and SCHEDD_CRON_NAME These variables will be honored through Condor versions 7.6, and support will be removed in Condor version 7.7. They are no longer documented as to their usage.

Defines a logical name to be used in the formation of related configuration macro names. This macro made other Daemon ClassAd Hook macros more readable and maintainable. A common example was

```
STARTD_CRON_NAME = HAWKEYE
```

This example allowed the naming of other related macros to contain the string HAWKEYE in their name, replacing the string STARTD_CRON.

The value of these variables may not be BENCHMARKS. The Daemon ClassAd Hook mechanism is used to implement a set of provided hooks that provide benchmark attributes.

STARTD_CRON_CONFIG_VAL and SCHEDD_CRON_CONFIG_VAL and BENCHMARKS_CONFIG_VAL This configuration variable can be used to specify the path and executable name of the *condor_config_val* program which the jobs (hooks) should use to get configuration information from the daemon. If defined, an environment variable by the same name with the same value will be passed to all jobs.

STARTD_CRON_AUTOPUBLISH Optional setting that determines if the *condor_startd* should automatically publish a new update to the *condor_collector* after any of the jobs produce output. Beware that enabling this setting can greatly increase the network traffic in a Condor pool, especially when many modules are executed, or if the period in which they run is short. There are three possible (case insensitive) values for this variable:

Never This default value causes the *condor_startd* to not automatically publish updates based on any jobs. Instead, updates rely on the usual behavior for sending updates, which is periodic, based on the UPDATE_INTERVAL configuration variable, or whenever a given slot changes state.

Always Causes the *condor_startd* to always send a new update to the *condor_collector* whenever any job exits.

If_Changed Causes the *condor_startd* to only send a new update to the *condor_collector* if the output produced by a given job is different than the previous output of the same job. The only exception is the LastUpdate attribute, which is automatically set for all jobs to be the timestamp when the job last ran. It is ignored when STARTD_CRON_AUTOPUBLISH is set to If_Changed.

STARTD_CRON_JOBLIST and SCHEDD_CRON_JOBLIST and BENCHMARKS_JOBLIST These configuration variables are defined by a comma and/or white space separated list of job names to run. Each is the logical name of a job. This name must be unique; no two jobs may have the same name.

STARTD_CRON_<JobName>_PREFIX and SCHEDD_CRON_<JobName>_PREFIX and BENCHMARKS_<JobName>_ Specifies a string which is prepended by Condor to all attribute names that the job generates. The use of prefixes avoids the conflicts that would be caused by attributes of the same name generated and utilized by different jobs. For example, if a module prefix is xyz_, and an individual attribute is named abc, then the resulting attribute name will be xyz_abc. Due to restrictions on ClassAd names, a prefix is only permitted to contain alpha-numeric characters and the underscore character.

<JobName> is the logical name assigned for a job as defined by configuration variable STARTD_CRON_JOBLIST, SCHEDD_CRON_JOBLIST, or BENCHMARKS_JOBLIST.

STARTD_CRON_<JobName>_SLOTS and BENCHMARKS_<JobName>_SLOTS A comma separated list of slots. The output of the job specified by <JobName> is incorporated into ClassAds; this list specifies which slots are to incorporate the output attributes of the job. If not specified, the default is to incorporate the output attributes into the ClassAd of all slots.

<JobName> is the logical name assigned for a job as defined by configuration variable STARTD_CRON_JOBLIST or BENCHMARKS_JOBLIST.

STARTD_CRON_<JobName>_EXECUTABLE and SCHEDD_CRON_<JobName>_EXECUTABLE and BENCHMARKS_<J
The full path and executable to run for this job. Note that multiple jobs may specify the same executable, although the jobs need to have different logical names.

<JobName> is the logical name assigned for a job as defined by configuration variable STARTD_CRON_JOBLIST, SCHEDD_CRON_JOBLIST, or BENCHMARKS_JOBLIST.

STARTD_CRON_<JobName>_PERIOD and SCHEDD_CRON_<JobName>_PERIOD and BENCHMARKS_<JobName>_
The period specifies time intervals at which the job should be run. For periodic jobs, this is the time interval that passes between starting the execution of the job. The value may be specified in seconds, minutes, or hours. Specify this time by appending the character s, m, or h to the value. As an example, 5m starts the execution of the job every five minutes. If no character is appended to the value, seconds are used as a default. In WaitForExit mode, the value has a different meaning: the period specifies the length of time after the job ceases execution and before it is restarted. The minimum valid value of the period is 1 second.

<JobName> is the logical name assigned for a job as defined by configuration variable STARTD_CRON_JOBLIST, SCHEDD_CRON_JOBLIST, or BENCHMARKS_JOBLIST.

STARTD_CRON_<JobName>_MODE and SCHEDD_CRON_<JobName>_MODE and BENCHMARKS_<JobName>_MODE
A string that specifies a mode within which the job operates. Legal values are

- Periodic, which is the default.
- WaitForExit
- OneShot
- OnDemand

<JobName> is the logical name assigned for a job as defined by configuration variable STARTD_CRON_JOBLIST, SCHEDD_CRON_JOBLIST, or BENCHMARKS_JOBLIST.

The default Periodic mode is used for most jobs. In this mode, the job is expected to be started by the *condor_startd* daemon, gather and publish its data, and then exit.

In WaitForExit mode the *condor_startd* daemon interprets the period as defined by STARTD_CRON_<JobName>_PERIOD differently. In this case, it refers to the amount of time to wait after the job exits before restarting it. With a value of 1, the job is kept running nearly continuously. In general, WaitForExit mode is for jobs that produce a periodic stream of updated data, but it can be used for other purposes, as well.

The OneShot mode is used for jobs that are run once at the start of the daemon. If the reconfig_rerun option is specified, the job will be run again after any reconfiguration.

The OnDemand mode is used only by the BENCHMARKS mechanism. All benchmark jobs must be be OnDemand jobs. Any other jobs specified as OnDemand will never run. Additional future features may allow for other OnDemand job uses.

STARTD_CRON_<JobName>_RECONFIG and SCHEDD_CRON_<JobName>_RECONFIG
A boolean value that when True, causes the daemon to send an HUP signal to the job when the daemon is reconfigured. The job is expected to reread its configuration at that time.

<JobName> is the logical name assigned for a job as defined by configuration variable STARTD_CRON_JOBLIST or SCHEDD_CRON_JOBLIST.

STARTD_CRON_<JobName>_RECONFIG_RERUN and SCHEDD_CRON_<JobName>_RECONFIG_RERUN
A boolean value that when True, causes the daemon ClassAd hooks mechanism to re-run the specified job when the daemon is reconfigured via *condor_reconfig*. The default value is False.

<JobName> is the logical name assigned for a job as defined by configuration variable STARTD_CRON_JOBLIST or SCHEDD_CRON_JOBLIST.

STARTD_CRON_<JobName>_JOB_LOAD and SCHEDD_CRON_<JobName>_JOB_LOAD and BENCHMARKS_<JobNa
A floating point value that represents the assumed and therefore expected CPU load that a job induces on the system. This job load is then used to limit the total number of jobs that run concurrently, by not starting new jobs if the assumed total load from all jobs is over a set threshold. The default value for each individual STARTD_CRON or a SCHEDD_CRON job is 0.01. The default value for each individual BENCHMARKS job is 1.0.

<JobName> is the logical name assigned for a job as defined by configuration variable STARTD_CRON_JOBLIST, SCHEDD_CRON_JOBLIST, or BENCHMARKS_JOBLIST.

STARTD_CRON_MAX_JOB_LOAD and SCHEDD_CRON_MAX_JOB_LOAD and BENCHMARKS_MAX_JOB_LOAD
A floating point value representing a threshold for CPU load, such that if starting another job would cause the sum of assumed loads for all running jobs to exceed this value, no further jobs will be started. The default value for STARTD_CRON or a SCHEDD_CRON hook managers is 0.1. This implies that a maximum of 10 jobs (using their default, assumed load) could be concurrently running. The default value for the BENCHMARKS hook manager is 1.0. This implies that only 1 BENCHMARKS job (at the default, assumed load) may be running.

STARTD_CRON_<JobName>_KILL and SCHEDD_CRON_<JobName>_KILL and BENCHMARKS_<JobName>_KILL
A boolean value applicable only for jobs with a MODE of anything other than WaitForExit. The default value is False.

This variable controls the behavior of the daemon hook manager when it detects that an instance of the job's executable is still running as it is time to invoke the job again. If True, the daemon hook manager will kill the currently running job and then invoke an new instance of the job. If False, the existing job invocation is allowed to continue running.

<JobName> is the logical name assigned for a job as defined by configuration variable STARTD_CRON_JOBLIST, SCHEDD_CRON_JOBLIST, or BENCHMARKS_JOBLIST.

STARTD_CRON_<JobName>_ARGS and SCHEDD_CRON_<JobName>_ARGS and BENCHMARKS_<JobName>_ARGS
The command line arguments to pass to the job as it is invoked. The first argument will be <JobName>.

<JobName> is the logical name assigned for a job as defined by configuration variable STARTD_CRON_JOBLIST, SCHEDD_CRON_JOBLIST, or BENCHMARKS_JOBLIST.

STARTD_CRON_<JobName>_ENV and SCHEDD_CRON_<JobName>_ENV and BENCHMARKS_<JobName>_ENV
The environment string to pass to the job. The syntax is the same as that of <DaemonName>_ENVIRONMENT as defined at 3.3.9.

<JobName> is the logical name assigned for a job as defined by configuration variable STARTD_CRON_JOBLIST, SCHEDD_CRON_JOBLIST, or BENCHMARKS_JOBLIST.

STARTD_CRON_<JobName>_CWD and SCHEDD_CRON_<JobName>_CWD and BENCHMARKS_<JobName>_CWD
The working directory in which to start the job.

<JobName> is the logical name assigned for a job as defined by configuration variable STARTD_CRON_JOBLIST, SCHEDD_CRON_JOBLIST, or BENCHMARKS_JOBLIST.

3.4 User Priorities and Negotiation

Condor uses priorities to determine machine allocation for jobs. This section details the priorities and the allocation of machines (negotiation).

For accounting purposes, each user is identified by username@uid_domain. Each user is assigned a priority value even if submitting jobs from different machines in the same domain, or even if submitting from multiple machines in the different domains.

The numerical priority value assigned to a user is inversely related to the *goodness* of the priority. A user with a numerical priority of 5 gets more resources than a user with a numerical priority of 50. There are two priority values assigned to Condor users:

- Real User Priority (RUP), which measures resource usage of the user.
- Effective User Priority (EUP), which determines the number of resources the user can get.

This section describes these two priorities and how they affect resource allocations in Condor. Documentation on configuring and controlling priorities may be found in section 3.3.17.

3.4.1 Real User Priority (RUP)

A user's RUP measures the resource usage of the user through time. Every user begins with a RUP of one half (0.5), and at steady state, the RUP of a user equilibrates to the number of resources used by that user. Therefore, if a specific user continuously uses exactly ten resources for a long period of time, the RUP of that user stabilizes at ten.

However, if the user decreases the number of resources used, the RUP gets better. The rate at which the priority value decays can be set by the macro PRIORITY_HALFLIFE , a time period defined in seconds. Intuitively, if the PRIORITY_HALFLIFE in a pool is set to 86400 (one day), and if a user whose RUP was 10 removes all his jobs, the user's RUP would be 5 one day later, 2.5 two days later, and so on.

3.4.2 Effective User Priority (EUP)

The effective user priority (EUP) of a user is used to determine how many resources that user may receive. The EUP is linearly related to the RUP by a *priority factor* which may be defined on a per-user basis. Unless otherwise configured, the priority factor for all users is 1.0, and so the EUP is the same as the the RUP. However, if desired, the priority factors of specific users (such as remote submitters) can be increased so that others are served preferentially.

The number of resources that a user may receive is inversely related to the ratio between the EUPs of submitting users. Therefore user A with EUP=5 will receive twice as many resources as user B with EUP=10 and four times as many resources as user C with EUP=20. However, if A does not use the full number of allocated resources, the available resources are repartitioned and distributed among remaining users according to the inverse ratio rule.

Condor supplies mechanisms to directly support two policies in which EUP may be useful:

Nice users A job may be submitted with the parameter `nice_user` set to TRUE in the submit command file. A nice user job gets its RUP boosted by the `NICE_USER_PRIO_FACTOR` priority factor specified in the configuration file, leading to a (usually very large) EUP. This corresponds to a low priority for resources. These jobs are therefore equivalent to Unix background jobs, which use resources not used by other Condor users.

Remote Users The flocking feature of Condor (see section 5.2) allows the *condor_schedd* to submit to more than one pool. In addition, the submit-only feature allows a user to run a *condor_schedd* that is submitting jobs into another pool. In such situations, submitters from other domains can submit to the local pool. It is often desirable to have Condor treat local users preferentially over these remote users. If configured, Condor will boost the RUPs of remote users by `REMOTE_PRIO_FACTOR` specified in the configuration file, thereby lowering their priority for resources.

The priority boost factors for individual users can be set with the **setfactor** option of *condor_userprio*. Details may be found in the *condor_userprio* manual page on page 873.

3.4.3 Priorities in Negotiation and Preemption

Priorities are used to ensure that users get their fair share of resources. The priority values are used at allocation time, meaning during negotiation and matchmaking. Therefore, there are ClassAd attributes that take on defined values only during negotiation, making them ephemeral. In addition to allocation, Condor may preempt a machine claim and reallocate it when conditions change.

Too many preemptions lead to thrashing, a condition in which negotiation for a machine identifies a new job with a better priority most every cycle. Each job is, in turn, preempted, and no job finishes. To avoid this situation, the `PREEMPTION_REQUIREMENTS` configuration variable is defined for and used only by the *condor_negotiator* daemon to specify the conditions that must be met for a preemption to occur. It is usually defined to deny preemption if a current running job has been

running for a relatively short period of time. This effectively limits the number of preemptions per resource per time interval. Note that PREEMPTION_REQUIREMENTS only applies to preemptions due to user priority. It does not have any effect if the machine's RANK expression prefers a different job, or if the machine's policy causes the job to vacate due to other activity on the machine. See section 3.5.9 for a general discussion of limiting preemption.

The following ephemeral attributes may be used within policy definitions. Care should be taken when using these attributes, due to their ephemeral nature; they are not always defined, so the usage of an expression to check if defined such as

```
(RemoteUserPrio =?= UNDEFINED)
```

is likely necessary.

Within these attributes, those with names that contain the string Submitter refer to characteristics about the candidate job's user; those with names that contain the string Remote refer to characteristics about the user currently using the resource. Further, those with names that end with the string ResourcesInUse have values that may change within the time period associated with a single negotiation cycle. Therefore, the configuration variables PREEMPTION_REQUIREMENTS_STABLE and and PREEMPTION_RANK_STABLE exist to inform the *condor_negotiator* daemon that values may change. See section 3.3.17 on page 231 for definitions of these configuration variables.

SubmitterUserPrio: A floating point value representing the user priority of the candidate job.

SubmitterUserResourcesInUse: The integer number of slots currently utilized by the user submitting the candidate job.

RemoteUserPrio: A floating point value representing the user priority of the job currently running on the machine. This version of the attribute, with no slot represented in the attribute name, refers to the current slot being evaluated.

Slot<N>_RemoteUserPrio: A floating point value representing the user priority of the job currently running on the particular slot represented by <N> on the machine.

RemoteUserResourcesInUse: The integer number of slots currently utilized by the user of the job currently running on the machine.

SubmitterGroupResourcesInUse: If the owner of the candidate job is a member of a valid accounting group, with a defined group quota, then this attribute is the integer number of slots currently utilized by the group.

SubmitterGroupQuota: If the owner of the candidate job is a member of a valid accounting group, with a defined group quota, then this attribute is the integer number of slots defined as the group's quota.

RemoteGroupResourcesInUse: If the owner of the currently running job is a member of a valid accounting group, with a defined group quota, then this attribute is the integer number of slots currently utilized by the group.

RemoteGroupQuota: If the owner of the currently running job is a member of a valid accounting group, with a defined group quota, then this attribute is the integer number of slots defined as the group's quota.

3.4.4 Priority Calculation

This section may be skipped if the reader so feels, but for the curious, here is Condor's priority calculation algorithm.

The RUP of a user u at time t, $\pi_r(u,t)$, is calculated every time interval δt using the formula

$$\pi_r(u,t) = \beta \times \pi(u, t-\delta t) + (1-\beta) \times \rho(u,t)$$

where $\rho(u,t)$ is the number of resources used by user u at time t, and $\beta = 0.5^{\delta t/h}$. h is the half life period set by `PRIORITY_HALFLIFE`.

The EUP of user u at time t, $\pi_e(u,t)$ is calculated by

$$\pi_e(u,t) = \pi_r(u,t) \times f(u,t)$$

where $f(u,t)$ is the priority boost factor for user u at time t.

As mentioned previously, the RUP calculation is designed so that at steady state, each user's RUP stabilizes at the number of resources used by that user. The definition of β ensures that the calculation of $\pi_r(u,t)$ can be calculated over non-uniform time intervals δt without affecting the calculation. The time interval δt varies due to events internal to the system, but Condor guarantees that unless the central manager machine is down, no matches will be unaccounted for due to this variance.

3.4.5 Negotiation

Negotiation is the method Condor undergoes periodically to match queued jobs with resources capable of running jobs. The *condor_negotiator* daemon is responsible for negotiation.

During a negotiation cycle, the *condor_negotiator* daemon accomplishes the following ordered list of items.

1. Build a list of all possible resources, regardless of the state of those resources.
2. Obtain a list of all job submitters (for the entire pool).
3. Sort the list of all job submitters based on EUP (see section 3.4.2 for an explanation of EUP). The submitter with the best priority is first within the sorted list.
4. Iterate until there are either no more resources to match, or no more jobs to match.

   For each submitter (in EUP order):

For each submitter, get each job. Since jobs may be submitted from more than one machine (hence to more than one *condor_schedd* daemon), here is a further definition of the ordering of these jobs. With jobs from a single *condor_schedd* daemon, jobs are typically returned in job priority order. When more than one *condor_schedd* daemon is involved, they are contacted in an undefined order. All jobs from a single *condor_schedd* daemon are considered before moving on to the next. For each job:

- For each machine in the pool that can execute jobs:

(a) If `machine.requirements` evaluates to `False` or `job.requirements` evaluates to `False`, skip this machine

(b) If the machine is in the Claimed state, but not running a job, skip this machine.

(c) If this machine is not running a job, add it to the potential match list by reason of No Preemption.

(d) If the machine is running a job

– If the `machine.RANK` on this job is better than the running job, add this machine to the potential match list by reason of Rank.

– If the EUP of this job is better than the EUP of the currently running job, and `PREEMPTION_REQUIREMENTS` is `True`, and the `machine.RANK` on this job is not worse than the currently running job, add this machine to the potential match list by reason of Priority.

- Of machines in the potential match list, sort by `NEGOTIATOR_PRE_JOB_RANK`, `job.RANK`, `NEGOTIATOR_POST_JOB_RANK`, Reason for claim (No Preemption, then Rank, then Priority), `PREEMPTION_RANK`
- The job is assigned to the top machine on the potential match list. The machine is removed from the list of resources to match (on this negotiation cycle).

The *condor_negotiator* asks the *condor_schedd* for the "next job" from a given submitter/user. Typically, the *condor_schedd* returns jobs in the order of job priority. If priorities are the same, job submission time is used; older jobs go first. If a cluster has multiple procs in it and one of the jobs cannot be matched, the *condor_schedd* will not return any more jobs in that cluster on that negotiation pass. This is an optimization based on the theory that the cluster jobs are similar. The configuration variable `NEGOTIATE_ALL_JOBS_IN_CLUSTER` disables the cluster-skipping optimization. Use of the configuration variable `SIGNIFICANT_ATTRIBUTES` will change the definition of what the *condor_schedd* considers a cluster from the default definition of all jobs that share the same `ClusterId`.

3.4.6 The Layperson's Description of the Pie Spin and Pie Slice

Condor schedules in a variety of ways. First, it takes all users who have submitted jobs and calculates their priority. Then, it totals the number of resources available at the moment, and using the ratios of the user priorities, it calculates the number of machines each user could get. This is their *pie slice*.

The Condor matchmaker goes in user priority order, contacts each user, and asks for job information. The *condor_schedd* daemon (on behalf of a user) tells the matchmaker about a job, and the matchmaker looks at available resources to create a list of resources that match the requirements expression. With the list of resources that match, it sorts them according to the rank expressions within ClassAds. If a machine prefers a job, the job is assigned to that machine, potentially preempting a job that might already be running on that machine. Otherwise, give the machine to the job that the job ranks highest. If the machine ranked highest is already running a job, we may preempt running job for the new job. A default policy for preemption states that the user must have a 20% better priority in order for preemption to succeed. If the job has no preferences as to what sort of machine it gets, matchmaking gives it the first idle resource to meet its requirements.

This matchmaking cycle continues until the user has received all of the machines in their pie slice. The matchmaker then contacts the next highest priority user and offers that user their pie slice worth of machines. After contacting all users, the cycle is repeated with any still available resources and recomputed pie slices. The matchmaker continues *spinning the pie* until it runs out of machines or all the *condor_schedd* daemons say they have no more jobs.

3.4.7 Group Accounting

By default, Condor does all accounting on a per-user basis, and this accounting is primarily used to compute priorities for Condor's fair-share scheduling algorithms. However, accounting can also be done on a per-group basis. Multiple users can all submit jobs into the same accounting group, and all of the jobs will be treated with the same priority.

To use an accounting group, each job inserts an attribute into the job ClassAd which defines the accounting group name for the job. A common name is decided upon and used for the group. The following line is an example that defines the attribute within the job's submit description file:

```
+AccountingGroup = "group_physics"
```

The `AccountingGroup` attribute is a string, and it therefore must be enclosed in double quote marks. The string may have a maximum length of 40 characters. The name should *not* be qualified with a domain. Certain parts of the Condor system do append the value `$(UID_DOMAIN)` (as specified in the configuration file on the submit machine) to this string for internal use. For example, if the value of `UID_DOMAIN` is `example.com`, and the accounting group name is as specified, *condor_userprio* will show statistics for this accounting group using the appended domain, for example

```
                                  Effective
User Name                         Priority
------------------------------    ---------
group_physics@example.com              0.50
user@example.com                      23.11
heavyuser@example.com                111.13
...
```

Additionally, the *condor_userprio* command allows administrators to remove an entity from the accounting system in Condor. The **-delete** option to *condor_userprio* accomplishes this if all the

jobs from a given accounting group are completed, and the administrator wishes to remove that group from the system. The **-delete** option identifies the accounting group with the fully-qualified name of the accounting group. For example

```
condor_userprio -delete group_physics@example.com
```

Condor removes entities itself as they are no longer relevant. Intervention by an administrator to delete entities can be beneficial when the use of thousands of short term accounting groups leads to scalability issues.

Note that the name of an accounting group may include a period (.). Inclusion of a period character in the accounting group name only has relevance if the portion of the name before the period matches a group name, as described in the next section on group quotas.

3.4.8 Hierarchical Group Quotas

The use of group quotas modifies the negotiation for available resources (machines) within a Condor pool. This solves the difficulties inherent when priorities assigned based on each single user are insufficient. This may be the case when different groups (of varying size) own computers, and the groups choose to combine their computers to form a Condor pool. Consider an imaginary Condor pool example with thirty computers; twenty computers are owned by the physics group and ten computers are owned by the chemistry group. One notion of fair allocation could be implemented by configuring the twenty machines owned by the physics group to prefer (using the `RANK` configuration macro) jobs submitted by the users identified as associated with the physics group. Likewise, the ten machines owned by the chemistry group are configured to prefer jobs from users associated with the the chemistry group. This routes jobs to execute on specific machines, perhaps causing more preemption than necessary. The (fair allocation) policy desired is likely somewhat different, if these thirty machines have been pooled. The desired policy does not tie users to specific sets of machines, but to numbers of machines (a quota). Given thirty similar machines, the desired policy allows users within the physics group to have preference on up to twenty of the machines within the pool, and the machines can be any of the machines that are available.

The implementation of quotas is hierarchical, such that quotas may be described for groups, subgroups, sub subgroups, etc. The hierarchy is described by adherence to a naming scheme set up in advance.

A quota for a set of users requires an identification of the set; members are called group users. Jobs under the group quota specify the group user with the `AccountingGroup` job ClassAd attribute. This is the same attribute as is used with group accounting.

The submit description file syntax for specifying a job is to be part of a group includes a series of names separated by the period character (`'.'`). Example syntax that shows only 2 levels of a (limited) hierarchy is

```
+AccountingGroup = "<group>.<subgroup>.<user>"
```

Both `<group>` and `<subgroup>` are names chosen for the group. Group names are case-insensitive for negotiation. The topmost level group name is not required to begin with the string `"group_"`, as in the examples `"group_physics.newton"` and `"group_chemistry.curie"`, but it is a useful convention, because group names must not conflict with subgroup or user names. Note that a job specifying a value for the `AccountingGroup` ClassAd attribute that lacks at least one period in the specification will cause the job to not be considered part of a group when negotiating, even if the group name (highest within the hierarchy) has a quota. Furthermore, there will be no warnings that the group quota is not in effect for the job, as this syntax defines group accounting.

Configuration controls the order of negotiation for groups, subgroups within the hierarchy defined, and individual users, as well as sets quotas (preferentially allocated numbers of machines) for the groups.

Quotas are categorized as either static or dynamic. A static quota specifies an integral numbers of machines (slots), independent of the size of the pool. A dynamic quota specifies a percentage of machines (slots) calculated based on the current number of machines in the pool. It is intended that only one of a static or a dynamic quota is defined for a specified group. If both are defined, then the static quota is implemented, and the dynamic quota is ignored.

Static Quotas In the hierarchical implementation, there are two cases defined here, to specify for the allocation of machines where there is both a group and a subgroup. In the first case, the sum for the numbers of machines within all of a group's subgroups totals to fewer than the specification for the group's static quota. For example:

```
GROUP_QUOTA_group_physics = 100
GROUP_QUOTA_group_physics.experiment1 = 20
GROUP_QUOTA_group_physics.experiment2 = 70
```

In this case, the unused, surplus quota of 10 machines is assigned to the higher level with in the hierarchy.

In the second case, the specification for the numbers of machines of a set of subgroups totals to more than the specification for the group's quota. For example:

```
GROUP_QUOTA_group_chemistry = 100
GROUP_QUOTA_group_chemistry.lab1 = 40
GROUP_QUOTA_group_chemistry.lab2 = 80
```

In this case, a warning is written to the log for the *condor_negotiator* daemon, and each of the subgroups will have their static quota scaled. In this example, the ratio 100/120 scales each subgroup. `lab1` will have a revised (floating point) quota of 33.333 machines, and `lab2` will have a revised (floating point) quota of 66.667 machines. As numbers of machines are always integer values, the floating point values are truncated for quota allocation. Fractional remainders resulting from the truncation are summed and assigned to the next higher level within the group hierarchy.

Dynamic Quotas A dynamic quota specifies a percentage of machines (slots) calculated based on the current number of machines in the pool. The quota is specified for a group (subgroup, etc.) by a floating point value in range 0.0 to 1.0 (inclusive).

Like static quota specification, there are two cases defined: when the dynamic quotas of all sub groups of a specific group sum to a fraction less than 1.0, and when the dynamic quotas of all sub groups of a specific group sum to greater than 1.0.

Here is an example configuration in which dynamic group quotas are assigned for a single group and its subgroups.

```
GROUP_QUOTA_DYNAMIC_group_econ = .6
GROUP_QUOTA_DYNAMIC_group_econ.project1 = .2
GROUP_QUOTA_DYNAMIC_group_econ.project2 = .15
GROUP_QUOTA_DYNAMIC_group_econ.project3 = .2
```

The sum of dynamic quotas for the subgroups is .55, which is less than the parent group, which has a dynamic quota of .6. If the pool has 100 slots, then the `project1` subgroup is assigned a quota that equals (100)(.6)(.2) = 12 machines. The `project2` subgroup is assigned a quota that equals (100)(.6)(.15) = 9 machines. The `project3` subgroup is assigned a quota that equals (100)(.6)(.2) = 12 machines. If the calculated dynamic quota of the subgroups resulted in non integer numbers of machines, integer numbers of machines are assigned based on the truncation of the non integer dynamic group quota. The unused, surplus quota of machines resulting from fractional remainders resulting from the truncation are summed and assigned to the next higher level within the group hierarchy.

Here is another example configuration in which dynamic group quotas are assigned for a single group and its subgroups.

```
GROUP_QUOTA_DYNAMIC_group_stat = .5
GROUP_QUOTA_DYNAMIC_group_stat.project1 = .2
GROUP_QUOTA_DYNAMIC_group_stat.project2 = .3
GROUP_QUOTA_DYNAMIC_group_stat.project3 = .2
```

In this case, the sum of dynamic quotas for the subgroups is .7, which is greater than the parent group, at .5 . A warning is written to the log for the *condor_negotiator* daemon, and each of the subgroups will have their dynamic group quota scaled by (.5)/(.7) = .7143 for this example. If the pool has 100 slots, then the `project1` subgroup is assigned a dynamic quota that equals (100)(.7143)(.2) which is 14.286 machines. The `project2` subgroup is assigned a dynamic quota that equals (100)(.7143)(.3) which is 21.429 machines. The `project3` subgroup is assigned a dynamic quota that equals (100)(.7143)(.2) which is 14.286 machines. The quota for each of `project1` and `project3` results in the truncated amount of 14 machines, with the 0.286 fractional portion of the calculated quota assigned to the parent group, `group_stat`. The quota for `project2` results in the truncated amount of 21 machines, with the 0.429 fractional portion of the calculated quota assigned to the parent group, `group_stat`.

Mixed Quotas - Both Static and Dynamic

This section has not yet been completed

3.5 Policy Configuration for the *condor_startd*

This section describes the configuration of machines, such that they, through the *condor_startd* daemon, implement a desired policy for when remote jobs should start, be suspended, (possibly) resumed, vacate (with a checkpoint) or be killed (no checkpoint). This policy is the heart of Condor's balancing act between the needs and wishes of resource owners (machine owners) and resource users (people submitting their jobs to Condor). Please read this section carefully if you plan to change any of the settings described here, as a wrong setting can have a severe impact on either the owners of machines in your pool (they may ask to be removed from the pool entirely) or the users of your pool (they may stop using Condor).

Before the details, there are a few things to note:

- Much of this section refers to ClassAd expressions. Please read through section 4.1 on ClassAd expressions before continuing.
- If defining the policy for an SMP machine (a multi-CPU machine), also read section 3.13.9 for specific information on configuring the *condor_startd* daemon for SMP machines. Each *slot* represented by the *condor_startd* daemon on an SMP machine has its own *state* and *activity* (as described below). In the future, each slot will be able to have its own individual policy expressions defined. Within this manual section, the word "machine" refers to an individual slot within an SMP machine.

To define a policy, set expressions in the configuration file (see section 3.3 on Configuring Condor for an introduction to Condor's configuration files). The expressions are evaluated in the context of the machine's ClassAd and a job ClassAd. The expressions can therefore reference attributes from either ClassAd. See the unnumbered Appendix on page 902 for a list of job ClassAd attributes. See the unnumbered Appendix on page 913 for a list of machine ClassAd attributes. The START expression is explained. It describes the conditions that must be met for a machine to start a job. The RANK expression for a machine is described. It allows the specification of the kinds of jobs a machine prefers to run. A final discussion details how the *condor_startd* daemon works. Included are the machine *states* and *activities*, to give an idea of what is possible in policy decisions. Two example policy settings are presented.

3.5.1 Startd ClassAd Attributes

The *condor_startd* daemon represents the machine on which it is running to the Condor pool. The daemon publishes characteristics about the machine in the machine's ClassAd to aid matchmaking with resource requests. The values of these attributes may be listed by using the command: *condor_status -l hostname*. On an SMP machine, the *condor_startd* will break the machine up and advertise it as separate slots, each with its own name and ClassAd.

3.5.2 The START expression

The most important expression to the *condor_startd* is the START expression. This expression describes the conditions that must be met for a machine to run a job. This expression can reference attributes in the machine's ClassAd (such as KeyboardIdle and LoadAvg) and attributes in a job ClassAd (such as Owner, Imagesize, and Cmd, the name of the executable the job will run). The value of the START expression plays a crucial role in determining the state and activity of a machine.

The Requirements expression is used for matching machines with jobs.

The *condor_startd* defines the Requirements expression by logically **and**ing the START expression and the IS_VALID_CHECKPOINT_PLATFORM expression.

In situations where a machine wants to make itself unavailable for further matches, the Requirements expression is set to FALSE. When the START expression locally evaluates to TRUE, the machine advertises the Requirements expression as TRUE and does not publish the START expression.

Normally, the expressions in the machine ClassAd are evaluated against certain request ClassAds in the *condor_negotiator* to see if there is a match, or against whatever request ClassAd currently has claimed the machine. However, by locally evaluating an expression, the machine only evaluates the expression against its own ClassAd. If an expression cannot be locally evaluated (because it references other expressions that are only found in a request ad, such as Owner or Imagesize), the expression is (usually) undefined. See section 4.1 for specifics on how undefined terms are handled in ClassAd expression evaluation.

A note of caution is in order when modifying the START to reference job ClassAd attributes. The default Is_OWNER expression is a function of the START expression

```
START =?= FALSE
```

See a detailed discussion of the IS_OWNER expression in section 3.5.7. However, the machine locally evaluates the IS_OWNER expression to determine if it is capable of running jobs for Condor. Any job ClassAd attributes appearing in the START expression, and hence in the IS_OWNER expression are undefined in this context, and may lead to unexpected behavior. Whenever the START expression is modified to reference job ClassAd attributes, the IS_OWNER expression should also be modified to reference only machine ClassAd attributes.

NOTE: If you have machines with lots of real memory and swap space such that the only scarce resource is CPU time, consider defining JOB_RENICE_INCREMENT so that Condor starts jobs on the machine with low priority. Then, further configure to set up the machines with:

```
START = True
SUSPEND = False
PREEMPT = False
KILL = False
```

In this way, Condor jobs always run and can never be kicked off from activity on the machine. However, because they would run with "nice priority", interactive response on the machines will not suffer. You probably would not notice Condor was running the jobs, assuming you had enough free memory for the Condor jobs that there was little swapping.

3.5.3 The IS_VALID_CHECKPOINT_PLATFORM expression

A checkpoint is the platform-dependent information necessary to continue the execution of a standard universe job. Therefore, the machine (platform) upon which a job executed and produced a checkpoint limits the machines (platforms) which may use the checkpoint to continue job execution. This platform-dependent information is no longer the obvious combination of architecture and operating system, but may include subtle items such as the difference between the normal, bigmem, and hugemem kernels within the Linux operating system. This results in the incorporation of a separate expression to indicate the ability of a machine to resume and continue the execution of a job that has produced a checkpoint. The REQUIREMENTS expression is dependent on this information.

At a high level, IS_VALID_CHECKPOINT_PLATFORM is an expression which becomes true when a job's checkpoint platform matches the current checkpointing platform of the machine. Since this expression is **and**ed with the START expression to produce the REQUIREMENTS expression, it must also behave correctly when evaluating in the context of jobs that are not standard universe.

In words, the current default policy for this expression:

Any non standard universe job may run on this machine. A standard universe job may run on machines with the new checkpointing identification system. A standard universe job may run if it has not yet produced a first checkpoint. If a standard universe job has produced a checkpoint, then make sure the checkpoint platforms between the job and the machine match.

The following is the default boolean expression for this policy. A JobUniverse value of 1 denotes the standard universe. This expression may be overridden in the Condor configuration files.

```
IS_VALID_CHECKPOINT_PLATFORM =
(
  ( (TARGET.JobUniverse == 1) == FALSE) ||

  (
    (MY.CheckpointPlatform =!= UNDEFINED) &&
    (
      (TARGET.LastCheckpointPlatform =?= MY.CheckpointPlatform) ||
      (TARGET.NumCkpts == 0)
    )
  )
)
```

IS_VALID_CHECKPOINT_PLATFORM is a separate policy expression because the complexity of IS_VALID_CHECKPOINT_PLATFORM can be very high. While this functionality is conceptually separate from the normal START policies usually constructed, it is also a part of the Requirements to allow the job to run.

3.5.4 The RANK expression

A machine may be configured to prefer certain jobs over others using the RANK expression. It is an expression, like any other in a machine ClassAd. It can reference any attribute found in either the machine ClassAd or a request ad (normally, in fact, it references things in the request ad). The most common use of this expression is likely to configure a machine to prefer to run jobs from the owner of that machine, or by extension, a group of machines to prefer jobs from the owners of those machines.

For example, imagine there is a small research group with 4 machines called tenorsax, piano, bass, and drums. These machines are owned by the 4 users coltrane, tyner, garrison, and jones, respectively.

Assume that there is a large Condor pool in your department, but you spent a lot of money on really fast machines for your group. You want to implement a policy that gives priority on your machines to anyone in your group. To achieve this, set the RANK expression on your machines to reference the Owner attribute and prefer requests where that attribute matches one of the people in your group as in

```
RANK = Owner == "coltrane" || Owner == "tyner" \
       || Owner == "garrison" || Owner == "jones"
```

The RANK expression is evaluated as a floating point number. However, like in C, boolean expressions evaluate to either 1 or 0 depending on if they are TRUE or FALSE. So, if this expression evaluated to 1 (because the remote job was owned by one of the preferred users), it would be a larger value than any other user (for whom the expression would evaluate to 0).

A more complex RANK expression has the same basic set up, where anyone from your group has priority on your machines. Its difference is that the machine owner has better priority on their own machine. To set this up for Jimmy Garrison, place the following entry in Jimmy Garrison's local configuration file `bass.local`:

```
RANK = (Owner == "coltrane") + (Owner == "tyner") \
       + ((Owner == "garrison") * 10) + (Owner == "jones")
```

NOTE: The parentheses in this expression are important, because "+" operator has higher default precedence than "==".

The use of "+" instead of "| | " allows us to distinguish which terms matched and which ones didn't. If anyone not in the John Coltrane quartet was running a job on the machine called bass, the RANK would evaluate numerically to 0, since none of the boolean terms evaluates to 1, and 0+0+0+0 still equals 0.

Suppose Elvin Jones submits a job. His job would match this machine (assuming the START was True for him at that time) and the RANK would numerically evaluate to 1. Therefore, Elvin would preempt the Condor job currently running. Assume that later Jimmy submits a job. The

RANK evaluates to 10, since the boolean that matches Jimmy gets multiplied by 10. Jimmy would preempt Elvin, and Jimmy's job would run on Jimmy's machine.

The RANK expression is not required to reference the `Owner` of the jobs. Perhaps there is one machine with an enormous amount of memory, and others with not much at all. You can configure your large-memory machine to prefer to run jobs with larger memory requirements:

```
RANK = ImageSize
```

That's all there is to it. The bigger the job, the more this machine wants to run it. It is an altruistic preference, always servicing the largest of jobs, no matter who submitted them. A little less altruistic is John's RANK that prefers his jobs over those with the largest `Imagesize`:

```
RANK = (Owner == "coltrane" * 1000000000000) + Imagesize
```

This RANK breaks if a job is submitted with an image size of more 10^{12} Kbytes. However, with that size, this RANK expression preferring that job would not be Condor's only problem!

3.5.5 Machine States

A machine is assigned a *state* by Condor. The state depends on whether or not the machine is available to run Condor jobs, and if so, what point in the negotiations has been reached. The possible states are

Owner The machine is being used by the machine owner, and/or is not available to run Condor jobs. When the machine first starts up, it begins in this state.

Unclaimed The machine is available to run Condor jobs, but it is not currently doing so.

Matched The machine is available to run jobs, and it has been matched by the negotiator with a specific schedd. That schedd just has not yet claimed this machine. In this state, the machine is unavailable for further matches.

Claimed The machine has been claimed by a schedd.

Preempting The machine was claimed by a schedd, but is now preempting that claim for one of the following reasons.

1. the owner of the machine came back
2. another user with higher priority has jobs waiting to run
3. another request that this resource would rather serve was found

Backfill The machine is running a backfill computation while waiting for either the machine owner to come back or to be matched with a Condor job. This state is only entered if the machine is specifically configured to enable backfill jobs.

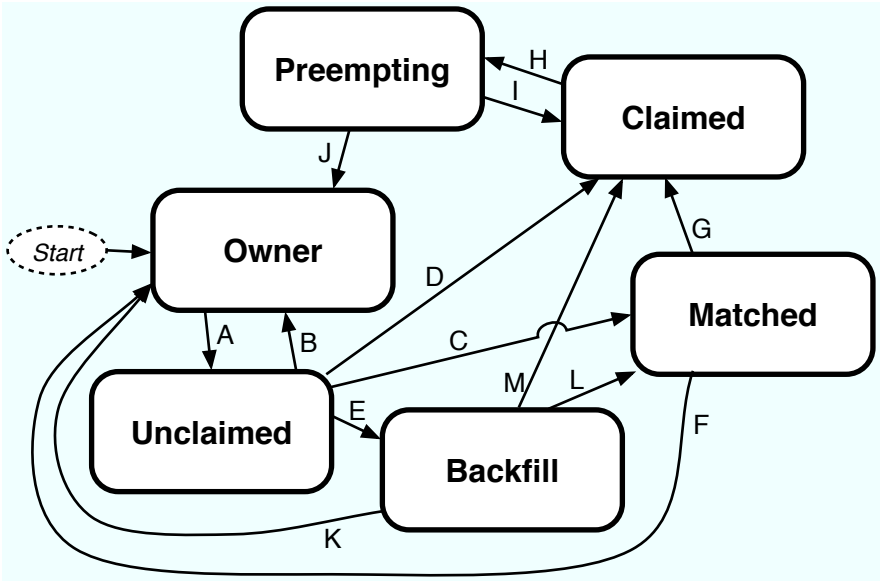

Figure 3.3: Machine States

Figure 3.3 shows the states and the possible transitions between the states.

Each transition is labeled with a letter. The cause of each transition is described below.

- Transitions out of the Owner state

  A The machine switches from Owner to Unclaimed whenever the `START` expression no longer locally evaluates to FALSE. This indicates that the machine is potentially available to run a Condor job.

- Transitions out of the Unclaimed state

  B The machine switches from Unclaimed back to Owner whenever the `START` expression locally evaluates to FALSE. This indicates that the machine is unavailable to run a Condor job and is in use by the resource owner.

  C The transition from Unclaimed to Matched happens whenever the *condor_negotiator* matches this resource with a Condor job.

  D The transition from Unclaimed directly to Claimed also happens if the *condor_negotiator* matches this resource with a Condor job. In this case the *condor_schedd* receives the match and initiates the claiming protocol with the machine before the *condor_startd* receives the match notification from the *condor_negotiator*.

  E The transition from Unclaimed to Backfill happens if the machine is configured to run backfill computations (see section 3.13.11) and the `START_BACKFILL` expression evaluates to TRUE.

- Transitions out of the Matched state

  F The machine moves from Matched to Owner if either the `START` expression locally evaluates to FALSE, or if the `MATCH_TIMEOUT` timer expires. This timeout is used to ensure that if a machine is matched with a given *condor_schedd*, but that *condor_schedd* does not contact the *condor_startd* to claim it, that the machine will give up on the match and become available to be matched again. In this case, since the `START` expression does not locally evaluate to FALSE, as soon as transition **F** is complete, the machine will immediately enter the Unclaimed state again (via transition **A**). The machine might also go from Matched to Owner if the *condor_schedd* attempts to perform the claiming protocol but encounters some sort of error. Finally, the machine will move into the Owner state if the *condor_startd* receives a *condor_vacate* command while it is in the Matched state.

  G The transition from Matched to Claimed occurs when the *condor_schedd* successfully completes the claiming protocol with the *condor_startd*.

- Transitions out of the Claimed state

  H From the Claimed state, the only possible destination is the Preempting state. This transition can be caused by many reasons:

  - The *condor_schedd* that has claimed the machine has no more work to perform and releases the claim
  - The `PREEMPT` expression evaluates to TRUE (which usually means the resource owner has started using the machine again and is now using the keyboard, mouse, CPU, etc)
  - The *condor_startd* receives a *condor_vacate* command
  - The *condor_startd* is told to shutdown (either via a signal or a *condor_off* command)
  - The resource is matched to a job with a better priority (either a better user priority, or one where the machine rank is higher)

- Transitions out of the Preempting state

  I The resource will move from Preempting back to Claimed if the resource was matched to a job with a better priority.

  J The resource will move from Preempting to Owner if the `PREEMPT` expression had evaluated to TRUE, if *condor_vacate* was used, or if the `START` expression locally evaluates to FALSE when the *condor_startd* has finished evicting whatever job it was running when it entered the Preempting state.

- Transitions out of the Backfill state

  K The resource will move from Backfill to Owner for the following reasons:

  - The `EVICT_BACKFILL` expression evaluates to TRUE
  - The *condor_startd* receives a *condor_vacate* command
  - The *condor_startd* is being shutdown

L The transition from Backfill to Matched occurs whenever a resource running a backfill computation is matched with a *condor_schedd* that wants to run a Condor job.

M The transition from Backfill directly to Claimed is similar to the transition from Unclaimed directly to Claimed. It only occurs if the *condor_schedd* completes the claiming protocol before the *condor_startd* receives the match notification from the *condor_negotiator*.

The Claimed State and Leases

When a *condor_schedd* claims a *condor_startd*, there is a claim lease. So long as the keep alive updates from the *condor_schedd* to the *condor_startd* continue to arrive, the lease is reset. If the lease duration passes with no updates, the *condor_startd* drops the claim and evicts any jobs the *condor_schedd* sent over.

The alive interval is the amount of time between, or the frequency at which the *condor_schedd* sends keep alive updates to all *condor_schedd* daemons. An alive update resets the claim lease at the *condor_startd*. Updates are UDP packets.

Initially, as when the *condor_schedd* starts up, the alive interval starts at the value set by the configuration variable `ALIVE_INTERVAL` . It may be modified when a job is started. The job's ClassAd attribute `JobLeaseDuration` is checked. If the value of `JobLeaseDuration/3` is less than the current alive interval, then the alive interval is set to either this lower value or the imposed lowest limit on the alive interval of 10 seconds. Thus, the alive interval starts at `ALIVE_INTERVAL` and goes down, never up.

If a claim lease expires, the *condor_startd* will drop the claim. The length of the claim lease is the job's ClassAd attribute `JobLeaseDuration`. `JobLeaseDuration` defaults to 20 minutes time, except when explicitly set within the job's submit description file. If `JobLeaseDuration` is explicitly set to 0, or it is not set as may be the case for a Web Services job that does not define the attribute, then `JobLeaseDuration` is given the Undefined value. Further, when undefined, the claim lease duration is calculated with `MAX_CLAIM_ALIVES_MISSED * alive interval`. The alive interval is the *current* value, as sent by the *condor_schedd*. If the *condor_schedd* reduces the current alive interval, it does not update the *condor_startd*.

3.5.6 Machine Activities

Within some machine states, *activities* of the machine are defined. The state has meaning regardless of activity. Differences between activities are significant. Therefore, a "state/activity" pair describes a machine. The following list describes all the possible state/activity pairs.

- Owner

  Idle This is the only activity for Owner state. As far as Condor is concerned the machine is Idle, since it is not doing anything for Condor.

- Unclaimed

  Idle This is the normal activity of Unclaimed machines. The machine is still Idle in that the machine owner is willing to let Condor jobs run, but Condor is not using the machine for anything.

  Benchmarking The machine is running benchmarks to determine the speed on this machine. This activity only occurs in the Unclaimed state. How often the activity occurs is determined by the `RUNBENCHMARKS` expression.

- Matched

  Idle When Matched, the machine is still Idle to Condor.

- Claimed

  Idle In this activity, the machine has been claimed, but the schedd that claimed it has yet to *activate* the claim by requesting a *condor_starter* to be spawned to service a job. The machine returns to this state (usually briefly) when jobs (and therefore *condor_starter*) finish.

  Busy Once a *condor_starter* has been started and the claim is active, the machine moves to the Busy activity to signify that it is doing something as far as Condor is concerned.

  Suspended If the job is suspended by Condor, the machine goes into the Suspended activity. The match between the schedd and machine has not been broken (the claim is still valid), but the job is not making any progress and Condor is no longer generating a load on the machine.

  Retiring When an active claim is about to be preempted for any reason, it enters retirement, while it waits for the current job to finish. The `MaxJobRetirementTime` expression determines how long to wait (counting since the time the job started). Once the job finishes or the retirement time expires, the Preempting state is entered.

- Preempting The preempting state is used for evicting a Condor job from a given machine. When the machine enters the Preempting state, it checks the `WANT_VACATE` expression to determine its activity.

  Vacating In the Vacating activity, the job that was running is in the process of checkpointing. As soon as the checkpoint process completes, the machine moves into either the Owner state or the Claimed state, depending on the reason for its preemption.

  Killing Killing means that the machine has requested the running job to exit the machine immediately, without checkpointing.

- Backfill

  Idle The machine is configured to run backfill jobs and is ready to do so, but it has not yet had a chance to spawn a backfill manager (for example, the BOINC client).

  Busy The machine is performing a backfill computation.

  Killing The machine was running a backfill computation, but it is now killing the job to either return resources to the machine owner, or to make room for a regular Condor job.

Figure 3.4 on page 293 gives the overall view of all machine states and activities and shows the possible transitions from one to another within the Condor system. Each transition is labeled with a number on the diagram, and transition numbers referred to in this manual will be **bold**.

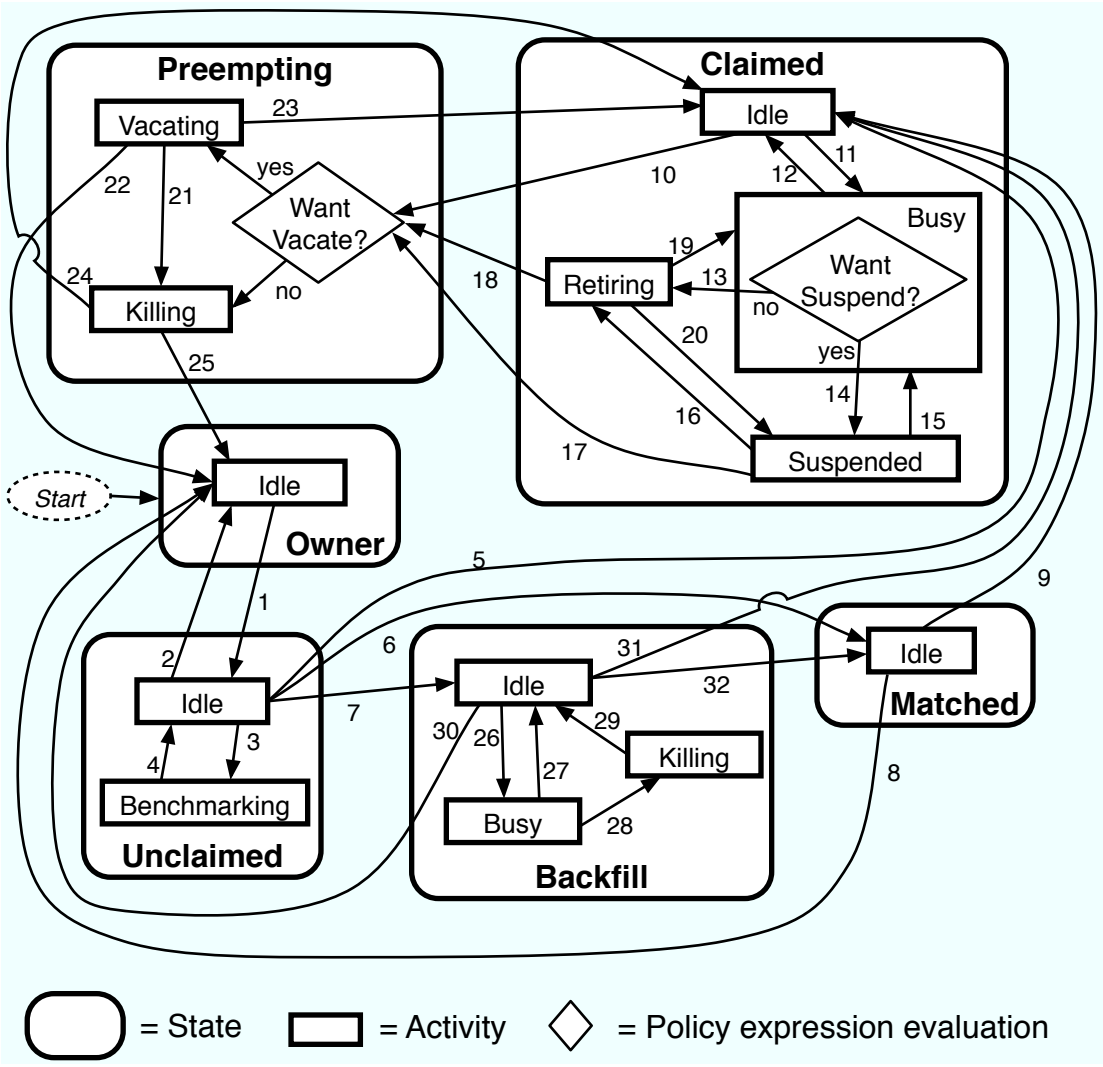

Figure 3.4: Machine States and Activities

Various expressions are used to determine when and if many of these state and activity transitions occur. Other transitions are initiated by parts of the Condor protocol (such as when the *condor_negotiator* matches a machine with a schedd). The following section describes the conditions that lead to the various state and activity transitions.

3.5.7 State and Activity Transitions

This section traces through all possible state and activity transitions within a machine and describes the conditions under which each one occurs. Whenever a transition occurs, Condor records when the machine entered its new activity and/or new state. These times are often used to write expressions that determine when further transitions occurred. For example, enter the Killing activity if a machine has been in the Vacating activity longer than a specified amount of time.

Owner State

When the startd is first spawned, the machine it represents enters the Owner state. The machine remains in the Owner state while the expression `IS_OWNER` is TRUE. If the `IS_OWNER` expression is FALSE, then the machine transitions to the Unclaimed state. The default value for the `IS_OWNER` expression is optimized for a shared resource

```
START =?= FALSE
```

So, the machine will remain in the Owner state as long as the `START` expression locally evaluates to FALSE. Section 3.5.2 provides more detail on the `START` expression. If the `START` locally evaluates to TRUE or cannot be locally evaluated (it evaluates to UNDEFINED), transition **1** occurs and the machine enters the Unclaimed state. The `IS_OWNER` expression is locally evaluated by the machine, and should not reference job ClassAd attributes, which would be UNDEFINED.

For dedicated resources, the recommended value for the `IS_OWNER` expression is FALSE.

The Owner state represents a resource that is in use by its interactive owner (for example, if the keyboard is being used). The Unclaimed state represents a resource that is neither in use by its interactive user, nor the Condor system. From Condor's point of view, there is little difference between the Owner and Unclaimed states. In both cases, the resource is not currently in use by the Condor system. However, if a job matches the resource's `START` expression, the resource is available to run a job, regardless of if it is in the Owner or Unclaimed state. The only differences between the two states are how the resource shows up in *condor_status* and other reporting tools, and the fact that Condor will not run benchmarking on a resource in the Owner state. As long as the `IS_OWNER` expression is TRUE, the machine is in the Owner State. When the `IS_OWNER` expression is FALSE, the machine goes into the Unclaimed State.

Here is an example that assumes that an `IS_OWNER` expression is not present in the configuration. If the `START` expression is

```
START = KeyboardIdle > 15 * $(MINUTE) && Owner == "coltrane"
```

and if `KeyboardIdle` is 34 seconds, then the machine would remain in the Owner state. Owner is undefined, and `anything && FALSE` is FALSE.

If, however, the `START` expression is

```
START = KeyboardIdle > 15 * $(MINUTE) || Owner == "coltrane"
```

and KeyboardIdle is 34 seconds, then the machine leaves the Owner state and becomes Unclaimed. This is because FALSE || UNDEFINED is UNDEFINED. So, while this machine is not available to just anybody, if user coltrane has jobs submitted, the machine is willing to run them. Any other user's jobs have to wait until KeyboardIdle exceeds 15 minutes. However, since coltrane might claim this resource, but has not yet, the machine goes to the Unclaimed state.

While in the Owner state, the startd polls the status of the machine every UPDATE_INTERVAL to see if anything has changed that would lead it to a different state. This minimizes the impact on the Owner while the Owner is using the machine. Frequently waking up, computing load averages, checking the access times on files, computing free swap space take time, and there is nothing time critical that the startd needs to be sure to notice as soon as it happens. If the START expression evaluates to TRUE and five minutes pass before the startd notices, that's a drop in the bucket of high-throughput computing.

The machine can only transition to the Unclaimed state from the Owner state. It does so when the IS_OWNER expression no longer evaluates to FALSE. By default, that happens when START no longer locally evaluates to FALSE.

Whenever the machine is not actively running a job, it will transition back to the Owner state if IS_OWNER evaluates to TRUE. Once a job is started, the value of IS_OWNER does not matter; the job either runs to completion or is preempted. Therefore, you must configure the preemption policy if you want to transition back to the Owner state from Claimed Busy.

Unclaimed State

If the IS_OWNER expression becomes TRUE, then the machine returns to the Owner state. If the IS_OWNER expression becomes FALSE, then the machine remains in the Unclaimed state. If the IS_OWNER expression is not present in the configuration files, then the default value for the IS_OWNER expression is

```
START =?= FALSE
```

so that while in the Unclaimed state, if the START expression locally evaluates to FALSE, the machine returns to the Owner state by transition **2**.

When in the Unclaimed state, the RUNBENCHMARKS expression is relevant. If RUNBENCHMARKS evaluates to TRUE while the machine is in the Unclaimed state, then the machine will transition from the Idle activity to the Benchmarking activity (transition **3**) and perform benchmarks to determine MIPS and KFLOPS. When the benchmarks complete, the machine returns to the Idle activity (transition **4**).

The startd automatically inserts an attribute, LastBenchmark, whenever it runs benchmarks, so commonly RunBenchmarks is defined in terms of this attribute, for example:

```
BenchmarkTimer = (CurrentTime - LastBenchmark)
RunBenchmarks = $(BenchmarkTimer) >= (4 * $(HOUR))
```

Here, a macro, `BenchmarkTimer` is defined to help write the expression. This macro holds the time since the last benchmark, so when this time exceeds 4 hours, we run the benchmarks again. The startd keeps a weighted average of these benchmarking results to try to get the most accurate numbers possible. This is why it is desirable for the startd to run them more than once in its lifetime.

NOTE: `LastBenchmark` is initialized to 0 before benchmarks have ever been run. To have the *condor_startd* run benchmarks as soon as the machine is Unclaimed (if it has not done so already), include a term using `LastBenchmark` as in the example above.

NOTE: If `RUNBENCHMARKS` is defined and set to something other than FALSE, the startd will automatically run one set of benchmarks when it first starts up. To disable benchmarks, both at startup and at any time thereafter, set `RUNBENCHMARKS` to FALSE or comment it out of the configuration file.

From the Unclaimed state, the machine can go to four other possible states: Owner (transition **2**), Backfill/Idle, Matched, or Claimed/Idle.

Once the *condor_negotiator* matches an Unclaimed machine with a requester at a given schedd, the negotiator sends a command to both parties, notifying them of the match. If the schedd receives that notification and initiates the claiming procedure with the machine before the negotiator's message gets to the machine, the Match state is skipped, and the machine goes directly to the Claimed/Idle state (transition **5**). However, normally the machine will enter the Matched state (transition **6**), even if it is only for a brief period of time.

If the machine has been configured to perform backfill jobs (see section 3.13.11), while it is in Unclaimed/Idle it will evaluate the `START_BACKFILL` expression. Once `START_BACKFILL` evaluates to TRUE, the machine will enter the Backfill/Idle state (transition **7**) to begin the process of running backfill jobs.

Matched State

The Matched state is not very interesting to Condor. Noteworthy in this state is that the machine lies about its `START` expression while in this state and says that `Requirements` are `False` to prevent being matched again before it has been claimed. Also interesting is that the startd starts a timer to make sure it does not stay in the Matched state too long. The timer is set with the `MATCH_TIMEOUT` configuration file macro. It is specified in seconds and defaults to 120 (2 minutes). If the schedd that was matched with this machine does not claim it within this period of time, the machine gives up, and goes back into the Owner state via transition **8**. It will probably leave the Owner state right away for the Unclaimed state again and wait for another match.

At any time while the machine is in the Matched state, if the `START` expression locally evaluates to FALSE, the machine enters the Owner state directly (transition **8**).

If the schedd that was matched with the machine claims it before the `MATCH_TIMEOUT` expires,

the machine goes into the Claimed/Idle state (transition **9**).

Claimed State

The Claimed state is certainly the most complex state. It has the most possible activities and the most expressions that determine its next activities. In addition, the *condor_checkpoint* and *condor_vacate* commands affect the machine when it is in the Claimed state. In general, there are two sets of expressions that might take effect. They depend on the universe of the request: standard or vanilla. The standard universe expressions are the normal expressions. For example:

```
WANT_SUSPEND            = True
WANT_VACATE             = $(ActivationTimer) > 10 * $(MINUTE)
SUSPEND                 = $(KeyboardBusy) || $(CPUBusy)
...
```

The vanilla expressions have the string"_VANILLA" appended to their names. For example:

```
WANT_SUSPEND_VANILLA    = True
WANT_VACATE_VANILLA     = True
SUSPEND_VANILLA         = $(KeyboardBusy) || $(CPUBusy)
...
```

Without specific vanilla versions, the normal versions will be used for all jobs, including vanilla jobs. In this manual, the normal expressions are referenced. The difference exists for the the resource owner that might want the machine to behave differently for vanilla jobs, since they cannot checkpoint. For example, owners may want vanilla jobs to remain suspended for longer than standard jobs.

While Claimed, the `POLLING_INTERVAL` takes effect, and the startd polls the machine much more frequently to evaluate its state.

If the machine owner starts typing on the console again, it is best to notice this as soon as possible to be able to start doing whatever the machine owner wants at that point. For SMP machines, if any slot is in the Claimed state, the startd polls the machine frequently. If already polling one slot, it does not cost much to evaluate the state of all the slots at the same time.

There are a variety of events that may cause the startd to try to get rid of or temporarily suspend a running job. Activity on the machine's console, load from other jobs, or shutdown of the startd via an administrative command are all possible sources of interference. Another one is the appearance of a higher priority claim to the machine by a different Condor user.

Depending on the configuration, the startd may respond quite differently to activity on the machine, such as keyboard activity or demand for the cpu from processes that are not managed by Condor. The startd can be configured to completely ignore such activity or to suspend the job or even to kill it. A standard configuration for a desktop machine might be to go through successive

levels of getting the job out of the way. The first and least costly to the job is suspending it. This works for both standard and vanilla jobs. If suspending the job for a short while does not satisfy the machine owner (the owner is still using the machine after a specific period of time), the startd moves on to vacating the job. Vacating a standard universe job involves performing a checkpoint so that the work already completed is not lost. Vanilla jobs are sent a *soft kill signal* so that they can gracefully shut down if necessary; the default is SIGTERM. If vacating does not satisfy the machine owner (usually because it is taking too long and the owner wants their machine back *now*), the final, most drastic stage is reached: killing. Killing is a quick death to the job, using a hard-kill signal that cannot be intercepted by the application. For vanilla jobs that do no special signal handling, vacating and killing are equivalent.

The WANT_SUSPEND expression determines if the machine will evaluate the SUSPEND expression to consider entering the Suspended activity. The WANT_VACATE expression determines what happens when the machine enters the Preempting state. It will go to the Vacating activity or directly to Killing. If one or both of these expressions evaluates to FALSE, the machine will skip that stage of getting rid of the job and proceed directly to the more drastic stages.

When the machine first enters the Claimed state, it goes to the Idle activity. From there, it has two options. It can enter the Preempting state via transition **10** (if a *condor_vacate* arrives, or if the START expression locally evaluates to FALSE), or it can enter the Busy activity (transition **11**) if the schedd that has claimed the machine decides to activate the claim and start a job.

From Claimed/Busy, the machine can transition to three other state/activity pairs. The startd evaluates the WANT_SUSPEND expression to decide which other expressions to evaluate. If WANT_SUSPEND is TRUE, then the startd evaluates the SUSPEND expression. If WANT_SUSPEND is any value other than TRUE, then the startd will evaluate the PREEMPT expression and skip the Suspended activity entirely. By transition, the possible state/activity destinations from Claimed/Busy:

Claimed/Idle If the starter that is serving a given job exits (for example because the jobs completes), the machine will go to Claimed/Idle (transition **12**).

Claimed/Retiring If WANT_SUSPEND is FALSE and the PREEMPT expression is TRUE, the machine enters the Retiring activity (transition **13**). From there, it waits for a configurable amount of time for the job to finish before moving on to preemption.

Another reason the machine would go from Claimed/Busy to Claimed/Retiring is if the *condor_negotiator* matched the machine with a "better" match. This better match could either be from the machine's perspective using the startd RANK expression, or it could be from the negotiator's perspective due to a job with a higher user priority.

Another case resulting in a transition to Claimed/Retiring is when the startd is being shut down. The only exception is a "fast" shutdown, which bypasses retirement completely.

Claimed/Suspended If both the WANT_SUSPEND and SUSPEND expressions evaluate to TRUE, the machine suspends the job (transition **14**).

If a *condor_checkpoint* command arrives, or the PERIODIC_CHECKPOINT expression evaluates to TRUE, there is no state change. The startd has no way of knowing when this process

completes, so periodic checkpointing can not be another state. Periodic checkpointing remains in the Claimed/Busy state and appears as a running job.

From the Claimed/Suspended state, the following transitions may occur:

Claimed/Busy If the `CONTINUE` expression evaluates to TRUE, the machine resumes the job and enters the Claimed/Busy state (transition **15**) or the Claimed/Retiring state (transition **16**), depending on whether the claim has been preempted.

Claimed/Retiring If the `PREEMPT` expression is TRUE, the machine will enter the Claimed/Retiring activity (transition **16**).

Preempting If the claim is in suspended retirement and the retirement time expires, the job enters the Preempting state (transition **17**). This is only possible if `MaxJobRetirementTime` *decreases* during the suspension.

For the Claimed/Retiring state, the following transitions may occur:

Preempting If the job finishes or the job's run time exceeds the value defined for the job ClassAd attribute `MaxJobRetirementTime`, the Preempting state is entered (transition **18**). The run time is computed from the time when the job was started by the startd minus any suspension time. When retiring due to *condor_startd* daemon shutdown or restart, it is possible for the administrator to issue a *peaceful* shutdown command, which causes `MaxJobRetirementTime` to effectively be infinite, avoiding any killing of jobs. It is also possible for the administrator to issue a *fast* shutdown command, which causes `MaxJobRetirementTime` to be effectively 0.

Claimed/Busy If the startd was retiring because of a preempting claim only and the preempting claim goes away, the normal Claimed/Busy state is resumed (transition **19**). If instead the retirement is due to owner activity (`PREEMPT`) or the startd is being shut down, no unretirement is possible.

Claimed/Suspended In exactly the same way that suspension may happen from the Claimed/Busy state, it may also happen during the Claimed/Retiring state (transition **20**). In this case, when the job continues from suspension, it moves back into Claimed/Retiring (transition **16**) instead of Claimed/Busy (transition **15**).

Preempting State

The Preempting state is less complex than the Claimed state. There are two activities. Depending on the value of `WANT_VACATE`, a machine will be in the Vacating activity (if TRUE) or the Killing activity (if FALSE).

While in the Preempting state (regardless of activity) the machine advertises its `Requirements` expression as FALSE to signify that it is not available for further matches, either because it is about to transition to the Owner state, or because it has already been matched with

one preempting match, and further preempting matches are disallowed until the machine has been claimed by the new match.

The main function of the Preempting state is to get rid of the starter associated with the resource. If the *condor_starter* associated with a given claim exits while the machine is still in the Vacating activity, then the job successfully completed a graceful shutdown. For standard universe jobs, this means that a checkpoint was saved. For other jobs, this means the application was given an opportunity to do a graceful shutdown, by intercepting the soft kill signal.

If the machine is in the Vacating activity, it keeps evaluating the `KILL` expression. As soon as this expression evaluates to TRUE, the machine enters the Killing activity (transition **21**).

When the starter exits, or if there was no starter running when the machine enters the Preempting state (transition **10**), the other purpose of the Preempting state is completed: notifying the schedd that had claimed this machine that the claim is broken.

At this point, the machine enters either the Owner state by transition **22** (if the job was preempted because the machine owner came back) or the Claimed/Idle state by transition **23** (if the job was preempted because a better match was found).

If the machine enters the Killing activity, (because either `WANT_VACATE` was FALSE or the `KILL` expression evaluated to TRUE), it attempts to force the *condor_starter* to immediately kill the underlying Condor job. Once the machine has begun to hard kill the Condor job, the *condor_startd* starts a timer, the length of which is defined by the `KILLING_TIMEOUT` macro. This macro is defined in seconds and defaults to 30. If this timer expires and the machine is still in the Killing activity, something has gone seriously wrong with the *condor_starter* and the startd tries to vacate the job immediately by sending SIGKILL to all of the *condor_starter*'s children, and then to the *condor_starter* itself.

Once the *condor_starter* has killed off all the processes associated with the job and exited, and once the schedd that had claimed the machine is notified that the claim is broken, the machine will leave the Preempting/Killing state. If the job was preempted because a better match was found, the machine will enter Claimed/Idle (transition **24**). If the preemption was caused by the machine owner (the `PREEMPT` expression evaluated to TRUE, *condor_vacate* was used, etc), the machine will enter the Owner state (transition **25**).

Backfill State

The Backfill state is used whenever the machine is performing low priority background tasks to keep itself busy. For more information about backfill support in Condor, see section 3.13.11 on page 456. This state is only used if the machine has been configured to enable backfill computation, if a specific backfill manager has been installed and configured, and if the machine is otherwise idle (not being used interactively or for regular Condor computations). If the machine meets all these requirements, and the `START_BACKFILL` expression evaluates to TRUE, the machine will move from the Unclaimed/Idle state to Backfill/Idle (transition **7**).

Once a machine is in Backfill/Idle, it will immediately attempt to spawn whatever backfill man-

ager it has been configured to use (currently, only the BOINC client is supported as a backfill manager in Condor). Once the BOINC client is running, the machine will enter Backfill/Busy (transition **26**) to indicate that it is now performing a backfill computation.

NOTE: On SMP machines, the *condor_startd* will only spawn a single instance of the BOINC client, even if multiple slots are available to run backfill jobs. Therefore, only the first machine to enter Backfill/Idle will cause a copy of the BOINC client to start running. If a given slot on an SMP enters the Backfill state and a BOINC client is already running under this *condor_startd*, the slot will immediately enter Backfill/Busy without waiting to spawn another copy of the BOINC client.

If the BOINC client ever exits on its own (which normally wouldn't happen), the machine will go back to Backfill/Idle (transition **27**) where it will immediately attempt to respawn the BOINC client (and return to Backfill/Busy via transition **26**).

As the BOINC client is running a backfill computation, a number of events can occur that will drive the machine out of the Backfill state. The machine can get matched or claimed for a Condor job, interactive users can start using the machine again, the machine might be evicted with *condor_vacate*, or the *condor_startd* might be shutdown. All of these events cause the *condor_startd* to kill the BOINC client and all its descendants, and enter the Backfill/Killing state (transition **28**).

Once the BOINC client and all its children have exited the system, the machine will enter the Backfill/Idle state to indicate that the BOINC client is now gone (transition **29**). As soon as it enters Backfill/Idle after the BOINC client exits, the machine will go into another state, depending on what caused the BOINC client to be killed in the first place.

If the `EVICT_BACKFILL` expression evaluates to TRUE while a machine is in Backfill/Busy, after the BOINC client is gone, the machine will go back into the Owner/Idle state (transition **30**). The machine will also return to the Owner/Idle state after the BOINC client exits if *condor_vacate* was used, or if the *condor_startd* is being shutdown.

When a machine running backfill jobs is matched with a requester that wants to run a Condor job, the machine will either enter the Matched state, or go directly into Claimed/Idle. As with the case of a machine in Unclaimed/Idle (described above), the *condor_negotiator* informs both the *condor_startd* and the *condor_schedd* of the match, and the exact state transitions at the machine depend on what order the various entities initiate communication with each other. If the *condor_schedd* is notified of the match and sends a request to claim the *condor_startd* before the *condor_negotiator* has a chance to notify the *condor_startd*, once the BOINC client exits, the machine will immediately enter Claimed/Idle (transition **31**). Normally, the notification from the *condor_negotiator* will reach the *condor_startd* before the *condor_schedd* attempts to claim it. In this case, once the BOINC client exits, the machine will enter Matched/Idle (transition **32**).

3.5.8 State/Activity Transition Expression Summary

This section is a summary of the information from the previous sections. It serves as a quick reference.

START When TRUE, the machine is willing to spawn a remote Condor job.

RUNBENCHMARKS While in the Unclaimed state, the machine will run benchmarks whenever TRUE.

MATCH_TIMEOUT If the machine has been in the Matched state longer than this value, it will transition to the Owner state.

WANT_SUSPEND If TRUE, the machine evaluates the SUSPEND expression to see if it should transition to the Suspended activity. If any value other than TRUE, the machine will look at the PREEMPT expression.

SUSPEND If WANT_SUSPEND is TRUE, and the machine is in the Claimed/Busy state, it enters the Suspended activity if SUSPEND is TRUE.

CONTINUE If the machine is in the Claimed/Suspended state, it enter the Busy activity if CONTINUE is TRUE.

PREEMPT If the machine is either in the Claimed/Suspended activity, or is in the Claimed/Busy activity and WANT_SUSPEND is FALSE, the machine enters the Claimed/Retiring state whenever PREEMPT is TRUE.

CLAIM_WORKLIFE If provided, this expression specifies the number of seconds during which a claim will continue accepting new jobs. Once this time expires, any existing job may continue to run as usual, but once it finishes or is preempted, the claim is closed. This may be useful if you want to force periodic renegotiation of resources without preemption having to occur. For example, if you have some low-priority jobs which should never be interrupted with kill signals, you could prevent them from being killed with MaxJobRetirementTime, but now high-priority jobs may have to wait in line when they match to a machine that is busy running one of these uninterruptible jobs. You can prevent the high-priority jobs from ever matching to such a machine by using a rank expression in the job or in the negotiator's rank expressions, but then the low-priority claim will never be interrupted; it can keep running more jobs. The solution is to use CLAIM_WORKLIFE to force the claim to stop running additional jobs after a certain amount of time. The default value for CLAIM_WORKLIFE is -1, which is treated as an infinite claim worklife, so claims may be held indefinitely (as long as they are not preempted and the schedd does not relinquish them, of course).

MAXJOBRETIREMENTTIME If the machine is in the Claimed/Retiring state, this expression specifies the maximum time (in seconds) that the *condor_startd* will wait for the job to finish naturally (without any kill signals from the *condor_startd*). The clock starts when the job is started and is paused during any suspension. The job may provide its own expression for MaxJobRetirementTime, but this can only be used to take *less* than the time granted by the *condor_startd*, never more. For convenience, standard universe and nice_user jobs are submitted with a default retirement time of 0, so they will never wait in retirement unless the user overrides the default.

Once the job finishes or if the retirement time expires, the machine enters the Preempting state.

This expression is evaluated in the context of the job ClassAd, so it may refer to attributes of the current job as well as machine attributes. The expression is continually re-evaluated

while the job is running, so it is possible, though unusual, to have an expression that changes over time. For example, if you want the retirement time to drop to 0 if an especially high priority job is waiting for the current job to retire, you could use `PreemptingRank` in the expression. Example:

```
MAXJOBRETIREMENTTIME = 3600 * ( \
  MY.PreemptingRank =?= UNDEFINED || \
  PreemptingRank < 600)
```

In this example, the retirement time is 3600 seconds, but if a job gets matched to this machine and it has a `PreemptingRank` of 600 or more, the retirement time drops to 0 and the current job is immediately preempted.

`WANT_VACATE` This is checked only when the `PREEMPT` expression is TRUE and the machine enters the Preempting state. If `WANT_VACATE` is TRUE, the machine enters the Vacating activity. If it is FALSE, the machine will proceed directly to the Killing activity.

`KILL` If the machine is in the Preempting/Vacating state, it enters Preempting/Killing whenever `KILL` is TRUE.

`KILLING_TIMEOUT` If the machine is in the Preempting/Killing state for longer than `KILLING_TIMEOUT` seconds, the *condor_startd* sends a SIGKILL to the *condor_starter* and all its children to try to kill the job as quickly as possible.

`PERIODIC_CHECKPOINT` If the machine is in the Claimed/Busy state and `PERIODIC_CHECKPOINT` is TRUE, the user's job begins a periodic checkpoint.

`RANK` If this expression evaluates to a higher number for a pending resource request than it does for the current request, the machine preempts the current request (enters the Preempting/Vacating state). When the preemption is complete, the machine enters the Claimed/Idle state with the new resource request claiming it.

`START_BACKFILL` When TRUE, if the machine is otherwise idle, it will enter the Backfill state and spawn a backfill computation (using BOINC).

`EVICT_BACKFILL` When TRUE, if the machine is currently running a backfill computation, it will kill the BOINC client and return to the Owner/Idle state.

3.5.9 Policy Settings

This section describes the default configuration policy and then provides examples of extensions to these policies.

Default Policy Settings

These settings are the default as shipped with Condor. They have been used for many years with no problems. The vanilla expressions are identical to the regular ones. (They are not listed here. If not defined, the standard expressions are used for vanilla jobs as well).

The following are macros to help write the expressions clearly.

StateTimer Amount of time in seconds in the current state.

ActivityTimer Amount of time in seconds in the current activity.

ActivationTimer Amount of time in seconds that the job has been running on this machine.

LastCkpt Amount of time since the last periodic checkpoint.

NonCondorLoadAvg The difference between the system load and the Condor load (the load generated by everything but Condor).

BackgroundLoad Amount of background load permitted on the machine and still start a Condor job.

HighLoad If the `$(NonCondorLoadAvg)` goes over this, the CPU is considered too busy, and eviction of the Condor job should start.

StartIdleTime Amount of time the keyboard must to be idle before Condor will start a job.

ContinueIdleTime Amount of time the keyboard must to be idle before resumption of a suspended job.

MaxSuspendTime Amount of time a job may be suspended before more drastic measures are taken.

MaxVacateTime Amount of time a job may be checkpointing before we give up and kill it outright.

KeyboardBusy A boolean expression that evaluates to TRUE when the keyboard is being used.

CPUIdle A boolean expression that evaluates to TRUE when the CPU is idle.

CPUBusy A boolean expression that evaluates to TRUE when the CPU is busy.

MachineBusy The CPU or the Keyboard is busy.

CPUIsBusy A boolean value set to the same value as `CPUBusy`.

CPUBusyTime The value 0 if `CPUBusy` is False; the time in seconds since `CPUBusy` became True.

```
##  These macros are here to help write legible expressions:
MINUTE          = 60
HOUR            = (60 * $(MINUTE))
StateTimer      = (CurrentTime - EnteredCurrentState)
ActivityTimer   = (CurrentTime - EnteredCurrentActivity)
ActivationTimer = (CurrentTime - JobStart)
LastCkpt        = (CurrentTime - LastPeriodicCheckpoint)

NonCondorLoadAvg        = (LoadAvg - CondorLoadAvg)
BackgroundLoad          = 0.3
HighLoad                = 0.5
StartIdleTime           = 15 * $(MINUTE)
ContinueIdleTime        = 5 * $(MINUTE)
MaxSuspendTime          = 10 * $(MINUTE)
MaxVacateTime           = 10 * $(MINUTE)

KeyboardBusy            = KeyboardIdle < $(MINUTE)
ConsoleBusy             = (ConsoleIdle  < $(MINUTE))
CPUIdle                 = $(NonCondorLoadAvg) <= $(BackgroundLoad)
CPUBusy                 = $(NonCondorLoadAvg) >= $(HighLoad)
KeyboardNotBusy         = ($(KeyboardBusy) == False)
MachineBusy             = ($(CPUBusy) || $(KeyboardBusy)
```

Macros are defined to want to suspend jobs (instead of killing them) in the case of jobs that use little memory, when the keyboard is not being used, and for vanilla universe jobs. We want to gracefully vacate jobs which have been running for more than 10 minutes or are vanilla universe jobs.

```
WANT_SUSPEND       = ( $(SmallJob) || $(KeyboardNotBusy) \
                       || $(IsVanilla) )
WANT_VACATE        = ( $(ActivationTimer) > 10 * $(MINUTE) \
                       || $(IsVanilla) )
```

Finally, definitions of the actual expressions. Start a job if the keyboard has been idle long enough and the load average is low enough OR the machine is currently running a Condor job. Note that Condor would only run one job at a time. It just may prefer to run a different job, as defined by the machine rank or user priorities.

```
START        = ( (KeyboardIdle > $(StartIdleTime)) \
                 && ( $(CPUIdle) || \
                      (State != "Unclaimed" && State != "Owner")) )
```

Suspend a job if the keyboard has been touched. Alternatively, suspend if the CPU has been busy for more than two minutes and the job has been running for more than 90 seconds.

```
SUSPEND         = ( $(KeyboardBusy) || \
                  ( (CpuBusyTime > 2 * $(MINUTE)) \
                     && $(ActivationTimer) > 90 ) )
```

Continue a suspended job if the CPU is idle, the Keyboard has been idle for long enough, and the job has been suspended more than 10 seconds.

```
CONTINUE        = ( $(CPUIdle) && ($(ActivityTimer) > 10) \
                  && (KeyboardIdle > $(ContinueIdleTime)) )
```

There are two conditions that signal preemption. The first condition is if the job is suspended, but it has been suspended too long. The second condition is if suspension is not desired and the machine is busy.

```
PREEMPT         = ( ((Activity == "Suspended") && \
                    ($(ActivityTimer) > $(MaxSuspendTime))) \
                    || (SUSPEND && (WANT_SUSPEND == False)) )
```

Do not give jobs any time to retire on their own when they are about to be preempted.

```
MAXJOBRETIREMENTTIME = 0
```

Kill jobs that take too long leaving gracefully.

```
KILL            = $(ActivityTimer) > $(MaxVacateTime)
```

Finally, specify periodic checkpointing. For jobs smaller than 60 Mbytes, do a periodic checkpoint every 6 hours. For larger jobs, only checkpoint every 12 hours.

```
PERIODIC_CHECKPOINT     = ( (ImageSize < 60000) && \
                            ($(LastCkpt) > (6 * $(HOUR))) ) || \
                          ( $(LastCkpt) > (12 * $(HOUR)) )
```

At UW-Madison, we have a fast network. We simplify our expression considerably to

```
PERIODIC_CHECKPOINT     = $(LastCkpt) > (3 * $(HOUR))
```

For reference, the entire set of policy settings are included once more without comments:

```
##  These macros are here to help write legible expressions:
MINUTE          = 60
```

```
HOUR            = (60 * $(MINUTE))
StateTimer      = (CurrentTime - EnteredCurrentState)
ActivityTimer   = (CurrentTime - EnteredCurrentActivity)
ActivationTimer = (CurrentTime - JobStart)
LastCkpt        = (CurrentTime - LastPeriodicCheckpoint)

NonCondorLoadAvg        = (LoadAvg - CondorLoadAvg)
BackgroundLoad          = 0.3
HighLoad                = 0.5
StartIdleTime           = 15 * $(MINUTE)
ContinueIdleTime        = 5 * $(MINUTE)
MaxSuspendTime          = 10 * $(MINUTE)
MaxVacateTime           = 10 * $(MINUTE)

KeyboardBusy            = KeyboardIdle < $(MINUTE)
ConsoleBusy             = (ConsoleIdle  < $(MINUTE))
CPUIdle                 = $(NonCondorLoadAvg) <= $(BackgroundLoad)
CPUBusy                 = $(NonCondorLoadAvg) >= $(HighLoad)
KeyboardNotBusy         = ($(KeyboardBusy) == False)
MachineBusy             = ($(CPUBusy) || $(KeyboardBusy)

WANT_SUSPEND       = ( $(SmallJob) || $(KeyboardNotBusy) \
                     || $(IsVanilla) )
WANT_VACATE        = ( $(ActivationTimer) > 10 * $(MINUTE) \
                     || $(IsVanilla) )
START        = ( (KeyboardIdle > $(StartIdleTime)) \
                  && ( $(CPUIdle) || \
                       (State != "Unclaimed" && State != "Owner")) )
SUSPEND         = ( $(KeyboardBusy) || \
                  ( (CpuBusyTime > 2 * $(MINUTE)) \
                     && $(ActivationTimer) > 90 ) )
CONTINUE        = ( $(CPUIdle) && ($(ActivityTimer) > 10) \
                  && (KeyboardIdle > $(ContinueIdleTime)) )
PREEMPT         = ( ((Activity == "Suspended") && \
                     ($(ActivityTimer) > $(MaxSuspendTime))) \
                     || (SUSPEND && (WANT_SUSPEND == False)) )
MAXJOBRETIREMENTTIME = 0
KILL            = $(ActivityTimer) > $(MaxVacateTime)
PERIODIC_CHECKPOINT     = ( (ImageSize < 60000) && \
                            ($(LastCkpt) > (6 * $(HOUR))) ) || \
                          ( $(LastCkpt) > (12 * $(HOUR)) )
```

Test-job Policy Example

This example shows how the default macros can be used to set up a machine for running test jobs from a specific user. Suppose we want the machine to behave normally, except if user coltrane submits a job. In that case, we want that job to start regardless of what is happening on the machine. We do not want the job suspended, vacated or killed. This is reasonable if we know coltrane is submitting very short running programs for testing purposes. The jobs should be executed right away. This works with any machine (or the whole pool, for that matter) by adding the following 5 expressions to the existing configuration:

```
START      = ($(START)) || Owner == "coltrane"
SUSPEND    = ($(SUSPEND)) && Owner != "coltrane"
CONTINUE   = $(CONTINUE)
PREEMPT    = ($(PREEMPT)) && Owner != "coltrane"
KILL       = $(KILL)
```

Notice that there is nothing special in either the `CONTINUE` or `KILL` expressions. If Coltrane's jobs never suspend, they never look at `CONTINUE`. Similarly, if they never preempt, they never look at `KILL`.

Time of Day Policy

Condor can be configured to only run jobs at certain times of the day. In general, we discourage configuring a system like this, since you can often get lots of good cycles out of machines, even when their owners say "I'm always using my machine during the day." However, if you submit mostly vanilla jobs or other jobs that cannot checkpoint, it might be a good idea to only allow the jobs to run when you know the machines will be idle and when they will not be interrupted.

To configure this kind of policy, you should use the `ClockMin` and `ClockDay` attributes, defined in section 3.5.1 on "Startd ClassAd Attributes". These are special attributes which are automatically inserted by the *condor_startd* into its ClassAd, so you can always reference them in your policy expressions. `ClockMin` defines the number of minutes that have passed since midnight. For example, 8:00am is 8 hours after midnight, or 8 * 60 minutes, or 480. 5:00pm is 17 hours after midnight, or 17 * 60, or 1020. `ClockDay` defines the day of the week, Sunday = 0, Monday = 1, and so on.

To make the policy expressions easy to read, we recommend using macros to define the time periods when you want jobs to run or not run. For example, assume regular "work hours" at your site are from 8:00am until 5:00pm, Monday through Friday:

```
WorkHours = ( (ClockMin >= 480 && ClockMin < 1020) && \
              (ClockDay > 0 && ClockDay < 6) )
AfterHours = ( (ClockMin < 480 || ClockMin >= 1020) || \
               (ClockDay == 0 || ClockDay == 6) )
```

Of course, you can fine-tune these settings by changing the definition of `AfterHours` and `WorkHours` for your site.

Assuming you are using the default policy expressions discussed above, there are only a few minor changes required to force Condor jobs to stay off of your machines during work hours:

```
# Only start jobs after hours.
START = $(AfterHours) && $(CPUIdle) && KeyboardIdle > $(StartIdleTime)

# Consider the machine busy during work hours, or if the keyboard or
# CPU are busy.
MachineBusy = ( $(WorkHours) || $(CPUBusy) || $(KeyboardBusy) )
```

By default, the `MachineBusy` macro is used to define the `SUSPEND` and `PREEMPT` expressions. If you have changed these expressions at your site, you will need to add `$(WorkHours)` to your `SUSPEND` and `PREEMPT` expressions as appropriate.

Depending on your site, you might also want to avoid suspending jobs during work hours, so that in the morning, if a job is running, it will be immediately preempted, instead of being suspended for some length of time:

```
WANT_SUSPEND = $(AfterHours)
```

Desktop/Non-Desktop Policy

Suppose you have two classes of machines in your pool: desktop machines and dedicated cluster machines. In this case, you might not want keyboard activity to have any effect on the dedicated machines. For example, when you log into these machines to debug some problem, you probably do not want a running job to suddenly be killed. Desktop machines, on the other hand, should do whatever is necessary to remain responsive to the user.

There are many ways to achieve the desired behavior. One way is to make a standard desktop policy and a standard non-desktop policy and to copy the desired one into the local configuration file for each machine. Another way is to define one standard policy (in condor_config) with a simple toggle that can be set in the local configuration file. The following example illustrates the latter approach.

For ease of use, an entire policy is included in this example. Some of the expressions are just the usual default settings.

```
# If "IsDesktop" is configured, make it an attribute of the machine ClassAd.
STARTD_ATTRS = IsDesktop

# Only consider starting jobs if:
```

```
# 1) the load average is low enough OR the machine is currently
#    running a Condor job
# 2) AND the user is not active (if a desktop)
START = ( ($(CPUIdle) || (State != "Unclaimed" && State != "Owner")) \
          && (IsDesktop =!= True || (KeyboardIdle > $(StartIdleTime))) )

# Suspend (instead of vacating/killing) for the following cases:
WANT_SUSPEND = ( $(SmallJob) || $(JustCpu) \
                 || $(IsVanilla) )

# When preempting, vacate (instead of killing) in the following cases:
WANT_VACATE  = ( $(ActivationTimer) > 10 * $(MINUTE) \
                 || $(IsVanilla) )

# Suspend jobs if:
# 1) The CPU has been busy for more than 2 minutes, AND
# 2) the job has been running for more than 90 seconds
# 3) OR suspend if this is a desktop and the user is active
SUSPEND = ( ((CpuBusyTime > 2 * $(MINUTE)) && ($(ActivationTimer) > 90)) \
            || ( IsDesktop =?= True && $(KeyboardBusy) ) )

# Continue jobs if:
# 1) the CPU is idle, AND
# 2) we've been suspended more than 5 minutes AND
# 3) the keyboard has been idle for long enough (if this is a desktop)
CONTINUE = ( $(CPUIdle) && ($(ActivityTimer) > 300) \
             && (IsDesktop =!= True || (KeyboardIdle > $(ContinueIdleTime))) )

# Preempt jobs if:
# 1) The job is suspended and has been suspended longer than we want
# 2) OR, we don't want to suspend this job, but the conditions to
#    suspend jobs have been met (someone is using the machine)
PREEMPT = ( ((Activity == "Suspended") && \
            ($(ActivityTimer) > $(MaxSuspendTime))) \
           || (SUSPEND && (WANT_SUSPEND == False)) )

# Replace 0 in the following expression with whatever amount of
# retirement time you want dedicated machines to provide.  The other part
# of the expression forces the whole expression to 0 on desktop
# machines.
MAXJOBRETIREMENTTIME = (IsDesktop =!= True) * 0

# Kill jobs if they have taken too long to vacate gracefully
KILL = $(ActivityTimer) > $(MaxVacateTime)
```

With this policy in condor_config, the local configuration files for desktops can be easily configured with the following line:

```
IsDesktop = True
```

In all other cases, the default policy described above will ignore keyboard activity.

Disabling Preemption

Preemption can result in jobs being killed by Condor. When this happens, the jobs remain in the queue and will be automatically rescheduled. We highly recommend designing jobs that work well in this environment, rather than simply disabling preemption.

Planning for preemption makes jobs more robust in the face of other sources of failure. One way to live happily with preemption is to use Condor's standard universe, which provides the ability to produce checkpoints. If a job is incompatible with the requirements of standard universe, the job can still gracefully shutdown and restart by intercepting the soft kill signal.

All that being said, there may be cases where it is appropriate to force Condor to never kill jobs within some upper time limit. This can be achieved with the following policy in the configuration of the execute nodes:

```
# When we want to kick a job off, let it run uninterrupted for
# up to 2 days before forcing it to vacate.
MAXJOBRETIREMENTTIME = $(HOUR) * 24 * 2
```

Construction of this expression may be more complicated. For example, it could provide a different retirement time to different users or different types of jobs. Also be aware that the job may come with its own definition of `MaxJobRetirementTime`, but this may only cause *less* retirement time to be used, never more than what the machine offers.

The longer the retirement time that is given, the slower reallocation of resources in the pool can become if there are long-running jobs. However, by preventing jobs from being killed, you may decrease the number of cycles that are wasted on non-checkpointable jobs that are killed. That is the basic trade off.

Note that the use of `MAXJOBRETIREMENTTIME` limits the killing of jobs, but it does not prevent the preemption of resource claims. Therefore, it is technically not a way of disabling preemption, but simply a way of forcing preempting claims to wait until an existing job finishes or runs out of time. In other words, it limits the preemption of jobs but not the preemption of claims.

Limiting the preemption of jobs is often more desirable than limiting the preemption of resource claims. However, if you really do want to limit the preemption of resource claims, the following policy may be used. Some of these settings apply to the execute node and some apply to the central manager, so this policy should be configured so that it is read by both.

```
#Disable preemption by machine activity.
PREEMPT = False
#Disable preemption by user priority.
PREEMPTION_REQUIREMENTS = False
#Disable preemption by machine RANK by ranking all jobs equally.
RANK = 0
#Since we are disabling claim preemption, we
# may as well optimize negotiation for this case:
NEGOTIATOR_CONSIDER_PREEMPTION = False
```

Be aware of the consequences of this policy. Without any preemption of resource claims, once the *condor_negotiator* gives the *condor_schedd* a match to a machine, the *condor_schedd* may hold onto this claim indefinitely, as long as the user keeps supplying more jobs to run. If this is not desired, force claims to be retired after some amount of time using `CLAIM_WORKLIFE` . This enforces a time limit, beyond which no new jobs may be started on an existing claim; therefore the *condor_schedd* daemon is forced to go back to the *condor_negotiator* to request a new match, if there is still more work to do. Example execute machine configuration to include in addition to the example above:

```
# after 20 minutes, schedd must renegotiate to run
# additional jobs on the machine
CLAIM_WORKLIFE = 1200
```

Also be aware that in all versions of Condor prior to 6.8.1, it is not advisable to set `NEGOTIATOR_CONSIDER_PREEMPTION` to False, because of a bug that can lead to some machines never being matched to jobs.

Job Suspension

As new jobs are submitted that receive a higher priority than currently executing jobs, the executing jobs may be preempted. If the preempted jobs are not capable of writing checkpoints, they lose whatever forward progress they have made, and are sent back to the job queue to await starting over again as another machine becomes available. An alternative to this is to use suspension to freeze the job while some other task runs, and then unfreeze it so that it can continue on from where it left off. This does not require any special handling in the job, unlike most strategies that take checkpoints. However, it does require a special configuration of Condor. This example implements a policy that allows the job to decide whether it should be evicted or suspended. The jobs announce their choice through the use of the invented job ClassAd attribute `IsSuspendableJob`, that is also utilized in the configuration.

The implementation of this policy utilizes two categories of slots, identified as suspendable or nonsuspendable. A job identifies which category of slot it wishes to run on. This affects two aspects of the policy:

- Of two jobs that might run on a slot, which job is chosen. The four cases that may occur depend on whether the currently running job identifies itself as suspendable or nonsuspendable, and whether the potentially running job identifies itself as suspendable or nonsuspendable.

1. If the currently running job is one that identifies itself as suspendable, and the potentially running job identifies itself as nonsuspendable, the currently running job is suspended, in favor of running the nonsuspendable one. This occurs independent of the user priority of the two jobs.
2. If both the currently running job and the potentially running job identify themselves as suspendable, then the relative priorities of the users and the preemption policy determines whether the new job will replace the existing job.
3. If both the currently running job and the potentially running job identify themselves as nonsuspendable, then the relative priorities of the users and the preemption policy determines whether the new job will replace the existing job.
4. If the currently running job is one that identifies itself as nonsuspendable, and the potentially running job identifies itself as suspendable, the currently running job continues running.

- What happens to a currently running job that is preempted. A job that identifies itself as suspendable will be suspended, which means it is frozen in place, and will later be unfrozen when the preempting job is finished. A job that identifies itself as nonsuspendable is evicted, which means it writes a checkpoint, when possible, and then is killed. The job will return to the idle state in the job queue, and it can try to run again in the future.

```
# Lie to Condor, to achieve 2 slots for each real slot
NUM_CPUS = $(DETECTED_CORES)*2
# There is no good way to tell Condor that the two slots should be treated
# as though they share the same real memory, so lie about how much
# memory we have.
MEMORY = $(DETECTED_MEMORY)*2

# Slots 1 through DETECTED_CORES are nonsuspendable and the rest are
# suspendable
IsSuspendableSlot = SlotID > $(DETECTED_CORES)

# If I am a suspendable slot, my corresponding nonsuspendable slot is
# my SlotID plus $(DETECTED_CORES)
NonSuspendableSlotState = eval(strcat("slot",SlotID-$(DETECTED_CORES),"_State")

# The above expression looks at slotX_State, so we need to add
# State to the list of slot attributes to advertise.
STARTD_SLOT_ATTRS = $(STARTD_SLOT_ATTRS) State

# For convenience, advertise these expressions in the machine ad.
STARTD_ATTRS = $(STARTD_ATTRS) IsSuspendableSlot NonSuspendableSlotState

MyNonSuspendableSlotIsIdle = \
  (NonSuspendableSlotState =!= "Claimed" && NonSuspendableSlotState =!= "Preempting")

# NonSuspendable slots are always willing to start jobs.
# Suspendable slots are only willing to start if the NonSuspendable slot is idle.
START = \
  IsSuspendableSlot!=True && IsSuspendableJob=!=True || \
  IsSuspendableSlot && IsSuspendableJob==True && $(MyNonSuspendableSlotIsIdle)
```

```
# Suspend the suspendable slot if the other slot is busy.
SUSPEND = \
  IsSuspendableSlot && $(MyNonSuspendableSlotIsIdle)!=True

WANT_SUSPEND = $(SUSPEND)

CONTINUE = ($(SUSPEND)) != True
```

Note that in this example, the job ClassAd attribute `IsSuspendableJob` has no special meaning to Condor. It is an invented name chosen for this example. To take advantage of the policy, a job that wishes to be suspended must submit the job so that this attribute is defined. The following line should be placed in the job's submit description file:

```
+IsSuspendableJob = True
```

3.6 Security

Security in Condor is a broad issue, with many aspects to consider. Because Condor's main purpose is to allow users to run arbitrary code on large numbers of computers, it is important to try to limit who can access a Condor pool and what privileges they have when using the pool. This section covers these topics.

There is a distinction between the kinds of resource attacks Condor can defeat, and the kinds of attacks Condor cannot defeat. Condor cannot prevent security breaches of users that can elevate their privilege to the root or administrator account. Condor does not run user jobs in sandboxes (standard universe jobs are a partial exception to this), so Condor cannot defeat all malicious actions by user jobs. An example of a malicious job is one that launches a distributed denial of service attack. Condor assumes that users are trustworthy. Condor can prevent unauthorized access to the Condor pool, to help ensure that only trusted users have access to the pool. In addition, Condor provides encryption and integrity checking, to ensure that data (both Condor's data and user jobs' data) has not been examined or tampered with while in transit.

Broadly speaking, the aspects of security in Condor may be categorized and described:

Users Authorization or capability in an operating system is based on a process owner. Both those that submit jobs and Condor daemons become process owners. The Condor system prefers that Condor daemons are run as the user `root`, while other common operations are owned by a user of Condor. Operations that do not belong to either `root` or a Condor user are often owned by the `condor` user. See Section 3.6.13 for more detail.

Authentication Proper identification of a user is accomplished by the process of authentication. It attempts to distinguish between real users and impostors. By default, Condor's authentication uses the user id (UID) to determine identity, but Condor can choose among a variety of authentication mechanisms, including the stronger authentication methods Kerberos and GSI.

Authorization Authorization specifies who is allowed to do what. Some users are allowed to submit jobs, while other users are allowed administrative privileges over Condor itself. Condor provides authorization on either a per-user or on a per-machine basis.

Privacy Condor may encrypt data sent across the network, which prevents others from viewing the data. With persistence and sufficient computing power, decryption is possible. Condor can encrypt the data sent for internal communication, as well as user data, such as files and executables. Encryption operates on network transmissions: unencrypted data is stored on disk.

Integrity The *man-in-the-middle* attack tampers with data without the awareness of either side of the communication. Condor's integrity check sends additional cryptographic data to verify that network data transmissions have not been tampered with. Note that the integrity information is only for network transmissions: data stored on disk does not have this integrity information.

3.6.1 Condor's Security Model

At the heart of Condor's security model is the notion that communications are subject to various security checks. A request from one Condor daemon to another may require authentication to prevent subversion of the system. A request from a user of Condor may need to be denied due to the confidential nature of the request. The security model handles these example situations and many more.

Requests to Condor are categorized into groups of *access levels*, based on the type of operation requested. The user of a specific request must be authorized at the required access level. For example, executing the *condor_status* command requires the `READ` access level. Actions that accomplish management tasks, such as shutting down or restarting of a daemon require an `ADMINISTRATOR` access level. See Section 3.6.7 for a full list of Condor's access levels and their meanings.

There are two sides to any communication or command invocation in Condor. One side is identified as the *client*, and the other side is identified as the *daemon*. The client is the party that initiates the command, and the daemon is the party that processes the command and responds. In some cases it is easy to distinguish the client from the daemon, while in other cases it is not as easy. Condor tools such as *condor_submit* and *condor_config_val* are clients. They send commands to daemons and act as clients in all their communications. For example, the *condor_submit* command communicates with the *condor_schedd*. Behind the scenes, Condor daemons also communicate with each other; in this case the daemon initiating the command plays the role of the client. For instance, the *condor_negotiator* daemon acts as a client when contacting the *condor_schedd* daemon to initiate matchmaking. Once a match has been found, the *condor_schedd* daemon acts as a client and contacts the *condor_startd* daemon.

Condor's security model is implemented using configuration. Commands in Condor are executed over TCP/IP network connections. While network communication enables Condor to manage resources that are distributed across an organization (or beyond), it also brings in security challenges. Condor must have ways of ensuring that commands are being sent by trustworthy users. Jobs that

are operating on sensitive data must be allowed to use encryption such that the data is not seen by outsiders. Jobs may need assurance that data has not been tampered with. These issues can be addressed with Condor's authentication, encryption, and integrity features.

Access Level Descriptions

Authorization is granted based on specified access levels. This list describes each access level, and provides examples of their usage. The levels implement a partial hierarchy; a higher level often implies a `READ` or both a `WRITE` and a `READ` level of access as described.

`READ` This access level can obtain or read information about Condor. Examples that require only `READ` access are viewing the status of the pool with *condor_status*, checking a job queue with *condor_q*, or viewing user priorities with *condor_userprio*. `READ` access does not allow any changes, and it does not allow job submission.

`WRITE` This access level is required to send (write) information to Condor. Examples that require `WRITE` access are job submission with *condor_submit* and advertising a machine so it appears in the pool (this is usually done automatically by the *condor_startd* daemon). The `WRITE` level of access implies `READ` access.

`ADMINISTRATOR` This access level has additional Condor administrator rights to the pool. It includes the ability to change user priorities (with the command *condor_userprio -set*), as well as the ability to turn Condor on and off (as with the commands *condor_on* and *condor_off*). The `ADMINISTRATOR` level of access implies both `READ` and `WRITE` access.

`SOAP` This access level is required for the authorization of any party that will use the Web Services (SOAP) interface to Condor. It is not a general access level to be used with the variety of configuration variables for authentication, encryption, and integrity checks.

`CONFIG` This access level is required to modify a daemon's configuration using the *condor_config_val* command. By default, this level of access can change any configuration parameters of a Condor pool, except those specified in the `condor_config.root` configuration file. The `CONFIG` level of access implies `READ` access.

`OWNER` This level of access is required for commands that the owner of a machine (any local user) should be able to use, in addition to the Condor administrators. An example that requires the `OWNER` access level is the *condor_vacate* command. The command causes the *condor_startd* daemon to vacate any Condor job currently running on a machine. The owner of that machine should be able to cause the removal of a job running on the machine.

`DAEMON` This access level is used for commands that are internal to the operation of Condor. An example of this internal operation is when the *condor_startd* daemon sends its ClassAd updates to the *condor_collector* daemon (which may be more specifically controlled by the `ADVERTISE_STARTD` access level). Authorization at this access level should only be given to the user account under which the Condor daemons run. The `DAEMON` level of access implies both `READ` and `WRITE` access. Any setting for this access level that is not defined will default to the corresponding setting in the `WRITE` access level.

NEGOTIATOR This access level is used specifically to verify that commands are sent by the *condor_negotiator* daemon. The *condor_negotiator* daemon runs on the central manager of the pool. Commands requiring this access level are the ones that tell the *condor_schedd* daemon to begin negotiating, and those that tell an available *condor_startd* daemon that it has been matched to a *condor_schedd* with jobs to run. The `NEGOTIATOR` level of access implies `READ` access.

ADVERTISE_MASTER This access level is used specifically for commands used to advertise a *condor_master* daemon to the collector. Any setting for this access level that is not defined will default to the corresponding setting in the `DAEMON` access level.

ADVERTISE_STARTD This access level is used specifically for commands used to advertise a *condor_startd* daemon to the collector. Any setting for this access level that is not defined will default to the corresponding setting in the `DAEMON` access level.

ADVERTISE_SCHEDD This access level is used specifically for commands used to advertise a *condor_schedd* daemon to the collector. Any setting for this access level that is not defined will default to the corresponding setting in the `DAEMON` access level.

CLIENT This access level is different from all the others. Whereas all of the other access levels refer to the security policy for accepting connections *from* others, the `CLIENT` access level applies when a Condor daemon or tool is connecting *to* some other Condor daemon. In other words, it specifies the policy of the client that is initiating the operation, rather than the server that is being contacted.

The following is a list of registered commands that daemons will accept. The list is ordered by daemon. For each daemon, the commands are grouped by the access level required for a daemon to accept the command from a given machine.

ALL DAEMONS:

WRITE The command sent as a result of *condor_reconfig* to reconfigure a daemon.

STARTD:

WRITE All commands that relate to a *condor_schedd* daemon claiming a machine, starting jobs there, or stopping those jobs.

The command that *condor_checkpoint* sends to periodically checkpoint all running jobs.

READ The command that *condor_preen* sends to request the current state of the *condor_startd* daemon.

OWNER The command that *condor_vacate* sends to cause any running jobs to stop running.

NEGOTIATOR The command that the *condor_negotiator* daemon sends to match a machine's *condor_startd* daemon with a given *condor_schedd* daemon.

NEGOTIATOR:

WRITE The command that initiates a new negotiation cycle. It is sent by the *condor_schedd* when new jobs are submitted or a *condor_reschedule* command is issued.

READ The command that can retrieve the current state of user priorities in the pool (sent by the *condor_userprio* command).

ADMINISTRATOR The command that can set the current values of user priorities (sent as a result of the `userprio -set` command).

COLLECTOR:

ADVERTISE_MASTER Commands that update the *condor_collector* daemon with new *condor_master* ClassAds.

ADVERTISE_SCHEDD Commands that update the *condor_collector* daemon with new *condor_schedd* ClassAds.

ADVERTISE_STARTD Commands that update the *condor_collector* daemon with new *condor_startd* ClassAds.

DAEMON All other commands that update the *condor_collector* daemon with new ClassAds. Note that the specific access levels such as `ADVERTISE_STARTD` default to the `DAEMON` settings, which in turn defaults to `WRITE`.

READ All commands that query the *condor_collector* daemon for ClassAds.

SCHEDD:

NEGOTIATOR The command that the *condor_negotiator* sends to begin negotiating with this *condor_schedd* to match its jobs with available *condor_startds*.

WRITE The command which *condor_reschedule* sends to the *condor_schedd* to get it to update the *condor_collector* with a current ClassAd and begin a negotiation cycle.

The commands which write information into the job queue (such as *condor_submit* and *condor_hold*). Note that for most commands which attempt to write to the job queue, Condor will perform an additional user-level authentication step. This additional user-level authentication prevents, for example, an ordinary user from removing a different user's jobs.

READ The command from any tool to view the status of the job queue.

The commands that a *condor_startd* sends to the *condor_schedd* when the *condor_schedd* daemon's claim is being preempted and also when the lease on the claim is renewed. These operations only require `READ` access, rather than `DAEMON` in order to limit the level of trust that the *condor_schedd* must have for the *condor_startd*. Success of these commands is only possible if the *condor_startd* knows the secret claim id, so effectively, authorization for these

commands is more specific than Condor's general security model implies. The *condor_schedd automatically* grants the *condor_startd* `READ` access for the duration of the claim. Therefore, if one desires to only authorize specific execute machines to run jobs, one must either limit which machines are allowed to advertise themselves to the pool (most common) or configure the *condor_schedd*'s `ALLOW_CLIENT` setting to only allow connections from the *condor_schedd* to the trusted execute machines.

MASTER: All commands are registered with `ADMINISTRATOR` access:

restart : Master restarts itself (and all its children)

off : Master shuts down all its children

off -master : Master shuts down all its children and exits

on : Master spawns all the daemons it is configured to spawn

3.6.2 Security Negotiation

Because of the wide range of environments and security demands necessary, Condor must be flexible. Configuration provides this flexibility. The process by which Condor determines the security settings that will be used when a connection is established is called *security negotiation*. Security negotiation's primary purpose is to determine which of the features of authentication, encryption, and integrity checking will be enabled for a connection. In addition, since Condor supports multiple technologies for authentication and encryption, security negotiation also determines which technology is chosen for the connection.

Security negotiation is a completely separate process from matchmaking, and should not be confused with any specific function of the *condor_negotiator* daemon. Security negotiation occurs when one Condor daemon or tool initiates communication with another Condor daemon, to determine the security settings by which the communication will be ruled. The *condor_negotiator* daemon does negotiation, whereby queued jobs and available machines within a pool go through the process of matchmaking (deciding out which machines will run which jobs).

Configuration

The configuration macro names that determine what features will be used during client-daemon communication follow the pattern:

```
SEC_<context>_<feature>
```

The `<feature>` portion of the macro name determines which security feature's policy is being set. `<feature>` may be any one of

```
AUTHENTICATION
ENCRYPTION
INTEGRITY
NEGOTIATION
```

The <context> component of the security policy macros can be used to craft a fine-grained security policy based on the type of communication taking place. <context> may be any one of

```
CLIENT
READ
WRITE
ADMINISTRATOR
CONFIG
OWNER
DAEMON
NEGOTIATOR
ADVERTISE_MASTER
ADVERTISE_STARTD
ADVERTISE_SCHEDD
DEFAULT
```

Any of these constructed configuration macros may be set to any of the following values:

```
REQUIRED
PREFERRED
OPTIONAL
NEVER
```

Security negotiation resolves various client-daemon combinations of desired security features in order to set a policy.

As an example, consider Frida the scientist. Frida wants to avoid authentication when possible. She sets

```
SEC_DEFAULT_AUTHENTICATION = OPTIONAL
```

The machine running the *condor_schedd* to which Frida will remotely submit jobs, however, is operated by a security-conscious system administrator who dutifully sets:

```
SEC_DEFAULT_AUTHENTICATION = REQUIRED
```

When Frida submits her jobs, Condor's security negotiation determines that authentication will be used, and allows the command to continue. This example illustrates the point that the most restrictive security policy sets the levels of security enforced. There is actually more to the understanding of

		Daemon Setting		
		NEVER	OPTIONAL	REQUIRED
Client Setting	NEVER	No	No	Fail
	REQUIRED	Fail	Yes	Yes

Table 3.1: Resolution of security negotiation.

		Daemon Setting			
		NEVER	OPTIONAL	PREFERRED	REQUIRED
Client Setting	NEVER	No	No	No	Fail
	OPTIONAL	No	No	Yes	Yes
	PREFERRED	No	Yes	Yes	Yes
	REQUIRED	Fail	Yes	Yes	Yes

Table 3.2: Resolution of security features.

this scenario. Some Condor commands, such as the use of *condor_submit* to submit jobs *always* require authentication of the submitter, no matter what the policy says. This is because the identity of the submitter needs to be known in order to carry out the operation. Others commands, such as *condor_q*, do not always require authentication, so in the above example, the server's policy would force Frida's *condor_q* queries to be authenticated, whereas a different policy could allow *condor_q* to happen without any authentication.

Whether or not security negotiation occurs depends on the setting at both the client and daemon side of the configuration variable(s) defined by `SEC_*_NEGOTIATION`. `SEC_DEFAULT_NEGOTIATION` is a variable representing the entire set of configuration variables for `NEGOTIATION`. For the client side setting, the only definitions that make sense are `REQUIRED` and `NEVER`. For the daemon side setting, the `PREFERRED` value makes no sense. Table 3.1 shows how security negotiation resolves various client-daemon combinations of security negotiation policy settings. Within the table, Yes means the security negotiation will take place. No means it will not. Fail means that the policy settings are incompatible and the communication cannot continue.

Enabling authentication, encryption, and integrity checks is dependent on security negotiation taking place. The enabled security negotiation further sets the policy for these other features. Table 3.2 shows how security features are resolved for client-daemon combinations of security feature policy settings. Like Table 3.1, Yes means the feature will be utilized. No means it will not. Fail implies incompatibility and the feature cannot be resolved.

The enabling of encryption and/or integrity checks is dependent on authentication taking place. The authentication provides a key exchange. The key is needed for both encryption and integrity checks.

Setting `SEC_CLIENT_<feature>` determines the policy for all outgoing commands. The policy for incoming commands (the daemon side of the communication) takes a more fine-grained approach that implements a set of access levels for the received command. For example, it is desirable to have all incoming administrative requests require authentication. Inquiries on pool status may not be so restrictive. To implement this, the administrator configures the policy:

```
SEC_ADMINISTRATOR_AUTHENTICATION = REQUIRED
SEC_READ_AUTHENTICATION          = OPTIONAL
```

The DEFAULT value for <context> provides a way to set a policy for all access levels (READ, WRITE, etc.) that do not have a specific configuration variable defined. In addition, some access levels will default to the settings specified for other access levels. For example, ADVERTISE_STARTD defaults to DAEMON, and DAEMON defaults to WRITE, which then defaults to the general DEFAULT setting.

Configuration for Security Methods

Authentication and encryption can each be accomplished by a variety of methods or technologies. Which method is utilized is determined during security negotiation.

The configuration macros that determine the methods to use for authentication and/or encryption are

```
SEC_<context>_AUTHENTICATION_METHODS
SEC_<context>_CRYPTO_METHODS
```

These macros are defined by a comma or space delimited list of possible methods to use. Section 3.6.3 lists all implemented authentication methods. Section 3.6.5 lists all implemented encryption methods.

3.6.3 Authentication

The client side of any communication uses one of two macros to specify whether authentication is to occur:

```
SEC_DEFAULT_AUTHENTICATION
SEC_CLIENT_AUTHENTICATION
```

For the daemon side, there are a larger number of macros to specify whether authentication is to take place, based upon the necessary access level:

```
SEC_DEFAULT_AUTHENTICATION
SEC_READ_AUTHENTICATION
SEC_WRITE_AUTHENTICATION
SEC_ADMINISTRATOR_AUTHENTICATION
SEC_CONFIG_AUTHENTICATION
SEC_OWNER_AUTHENTICATION
SEC_DAEMON_AUTHENTICATION
```

```
SEC_NEGOTIATOR_AUTHENTICATION
SEC_ADVERTISE_MASTER_AUTHENTICATION
SEC_ADVERTISE_STARTD_AUTHENTICATION
SEC_ADVERTISE_SCHEDD_AUTHENTICATION
```

As an example, the macro defined in the configuration file for a daemon as

```
SEC_WRITE_AUTHENTICATION = REQUIRED
```

signifies that the daemon must authenticate the client for any communication that requires the `WRITE` access level. If the daemon's configuration contains

```
SEC_DEFAULT_AUTHENTICATION = REQUIRED
```

and does not contain any other security configuration for `AUTHENTICATION`, then this default defines the daemon's needs for authentication over all access levels. Where a specific macro is defined, the more specific value takes precedence over the default definition.

If authentication is to be done, then the communicating parties must negotiate a mutually acceptable method of authentication to be used. A list of acceptable methods may be provided by the client, using the macros

```
SEC_DEFAULT_AUTHENTICATION_METHODS
SEC_CLIENT_AUTHENTICATION_METHODS
```

A list of acceptable methods may be provided by the daemon, using the macros

```
SEC_DEFAULT_AUTHENTICATION_METHODS
SEC_READ_AUTHENTICATION_METHODS
SEC_WRITE_AUTHENTICATION_METHODS
SEC_ADMINISTRATOR_AUTHENTICATION_METHODS
SEC_CONFIG_AUTHENTICATION_METHODS
SEC_OWNER_AUTHENTICATION_METHODS
SEC_DAEMON_AUTHENTICATION_METHODS
SEC_NEGOTIATOR_AUTHENTICATION_METHODS
SEC_ADVERTISE_MASTER_AUTHENTICATION_METHODS
SEC_ADVERTISE_STARTD_AUTHENTICATION_METHODS
SEC_ADVERTISE_SCHEDD_AUTHENTICATION_METHODS
```

The methods are given as a comma-separated list of acceptable values. These variables list the authentication methods that are available to be used. The ordering of the list defines preference; the first item in the list indicates the highest preference. Defined values are

```
GSI
```

```
SSL
KERBEROS
PASSWORD
FS
FS_REMOTE
NTSSPI
CLAIMTOBE
ANONYMOUS
```

For example, a client may be configured with:

```
SEC_CLIENT_AUTHENTICATION_METHODS = FS, GSI
```

and a daemon the client is trying to contact with:

```
SEC_DEFAULT_AUTHENTICATION_METHODS = GSI
```

Security negotiation will determine that GSI authentication is the only compatible choice. If there are multiple compatible authentication methods, security negotiation will make a list of acceptable methods and they will be tried in order until one succeeds.

As another example, the macro

```
SEC_DEFAULT_AUTHENTICATION_METHODS = KERBEROS, NTSSPI
```

indicates that either Kerberos or Windows authentication may be used, but Kerberos is preferred over Windows. Note that if the client and daemon agree that multiple authentication methods may be used, then they are tried in turn. For instance, if they both agree that Kerberos or NTSSPI may be used, then Kerberos will be tried first, and if there is a failure for any reason, then NTSSPI will be tried.

An additional specialized method of authentication exists for communication between the *condor_schedd* and *condor_startd*. It is especially useful when operating at large scale over high latency networks or in situations where it is inconvenient to set up one of the other methods of strong authentication between the submit and execute daemons. See the description of SEC_ENABLE_MATCH_PASSWORD_AUTHENTICATION on 254 for details.

If the configuration for a machine does not define any variable for SEC_<access-level>_AUTHENTICATION, then Condor uses a default value of OPTIONAL. Authentication will be required for any operation which modifies the job queue, such as *condor_qedit* and *condor_rm*. If the configuration for a machine does not define any variable for SEC_<access-level>_AUTHENTICATION_METHODS, the default value for a Unix machine is FS, KERBEROS, GSI. This default value for a Windows machine is NTSSPI, KERBEROS, GSI.

GSI Authentication

The GSI (Grid Security Infrastructure) protocol provides an avenue for Condor to do PKI-based (Public Key Infrastructure) authentication using X.509 certificates. The basics of GSI are well-documented elsewhere, such as http://www.globus.org/.

A simple introduction to this type of authentication defines Condor's use of terminology, and it illuminates the needed items that Condor must access to do this authentication. Assume that A authenticates to B. In this example, A is the client, and B is the daemon within their communication. This example's one-way authentication implies that B is verifying the identity of A, using the certificate A provides, and utilizing B's own set of trusted CAs (Certification Authorities). Client A provides its certificate (or proxy) to daemon B. B does two things: B checks that the certificate is valid, and B checks to see that the CA that signed A's certificate is one that B trusts.

For the GSI authentication protocol, an X.509 certificate is required. Files with predetermined names hold a certificate, a key, and optionally, a proxy. A separate directory has one or more files that become the list of trusted CAs.

Allowing Condor to do this GSI authentication requires knowledge of the locations of the client A's certificate and the daemon B's list of trusted CAs. When one side of the communication (as either client A or daemon B) is a Condor daemon, these locations are determined by configuration or by default locations. When one side of the communication (as a client A) is a user of Condor (the process owner of a Condor tool, for example *condor_submit*), these locations are determined by the pre-set values of environment variables or by default locations.

GSI certificate locations for Condor daemons For a Condor daemon, the certificate may be a single host certificate, and all Condor daemons on the same machine may share the same certificate. In some cases, the certificate can also be copied to other machines, where local copies are necessary. This may occur only in cases where a single host certificate can match multiple host names, something that is beyond the scope of this manual. The certificates must be protected by access rights to files, since the password file is not encrypted.

The specification of the location of the necessary files through configuration uses the following precedence.

1. Configuration variable `GSI_DAEMON_DIRECTORY` gives the complete path name to the directory that contains the certificate, key, and directory with trusted CAs. Condor uses this directory as follows in its construction of the following configuration variables:

   ```
   GSI_DAEMON_CERT           = $(GSI_DAEMON_DIRECTORY)/hostcert.pem
   GSI_DAEMON_KEY            = $(GSI_DAEMON_DIRECTORY)/hostkey.pem
   GSI_DAEMON_TRUSTED_CA_DIR = $(GSI_DAEMON_DIRECTORY)/certificates
   ```

   Note that no proxy is assumed in this case.

2. If the `GSI_DAEMON_DIRECTORY` is not defined, or when defined, the location may be overridden with specific configuration variables that specify the complete path and file name of the certificate with

   `GSI_DAEMON_CERT`

   the key with

GSI_DAEMON_KEY

a proxy with

GSI_DAEMON_PROXY

the complete path to the directory containing the list of trusted CAs with

GSI_DAEMON_TRUSTED_CA_DIR

3. The default location assumed is /etc/grid-security. Note that this implemented by setting the value of GSI_DAEMON_DIRECTORY.

When a daemon acts as the client within authentication, the daemon needs a listing of those from which it will accept certificates. This is done with GSI_DAEMON_NAME. This name is specified with the following format

```
GSI_DAEMON_NAME = /X.509/name/of/server/1,/X.509/name/of/server/2,...
```

Condor will also need a way to map an X.509 distinguished name to a Condor user id. There are two ways to accomplish this mapping. For a first way to specify the mapping, see section 3.6.4 to use Condor's unified map file. The second way to do the mapping is within an administrator-maintained GSI-specific file called an X.509 map file, mapping from X.509 Distinguished Name (DN) to Condor user id. It is similar to a Globus grid map file, except that it is only used for mapping to a user id, not for authorization. If the user names in the map file do not specify a domain for the user (specification would appear as user@domain), then the value of UID_DOMAIN is used. Entries (lines) in the file each contain two items. The first item in an entry is the X.509 certificate subject name, and it is enclosed in double quote marks (using the character "). The second item is the Condor user id. The two items in an entry are separated by tab or space character(s). Here is an example of an entry in an X.509 map file. Entries must be on a single line; this example is broken onto two lines for formatting reasons.

```
"/C=US/O=Globus/O=University of Wisconsin/
OU=Computer Sciences Department/CN=Alice Smith" asmith
```

Condor finds the map file in one of three ways. If the configuration variable GRIDMAP is defined, it gives the full path name to the map file. When not defined, Condor looks for the map file in

```
$(GSI_DAEMON_DIRECTORY)/grid-mapfile
```

If GSI_DAEMON_DIRECTORY is not defined, then the third place Condor looks for the map file is given by

```
/etc/grid-security/grid-mapfile
```

GSI certificate locations for Users The user specifies the location of a certificate, proxy, etc. in one of two ways:

1. Environment variables give the location of necessary items.
   X509_USER_PROXY gives the path and file name of the proxy. This proxy will have been created using the *grid-proxy-init* program, which will place the proxy in the /tmp directory with the file name being determined by the format:

```
/tmp/x509up_uXXXX
```

The specific file name is given by substituting the XXXX characters with the UID of the user. Note that when a valid proxy is used, the certificate and key locations are not needed.

X509_USER_CERT gives the path and file name of the certificate. It is also used if a proxy location has been checked, but the proxy is no longer valid.

X509_USER_KEY gives the path and file name of the key. Note that most keys are password encrypted, such that knowing the location could not lead to using the key.

X509_CERT_DIR gives the path to the directory containing the list of trusted CAs.

2. Without environment variables to give locations of necessary certificate information, Condor uses a default directory for the user. This directory is given by

   ```
   $(HOME)/.globus
   ```

Example GSI Security Configuration Here is an example portion of the configuration file that would enable and require GSI authentication, along with a minimal set of other variables to make it work.

```
SEC_DEFAULT_AUTHENTICATION = REQUIRED
SEC_DEFAULT_AUTHENTICATION_METHODS = GSI
SEC_DEFAULT_INTEGRITY = REQUIRED
GSI_DAEMON_DIRECTORY = /etc/grid-security
GRIDMAP = /etc/grid-security/grid-mapfile

# authorize based on user names produced by the map file
ALLOW_READ = *@cs.wisc.edu/*.cs.wisc.edu
ALLOW_DAEMON = condor@cs.wisc.edu/*.cs.wisc.edu
ALLOW_NEGOTIATOR = condor@cs.wisc.edu/condor.cs.wisc.edu, \
                   condor@cs.wisc.edu/condor2.cs.wisc.edu
ALLOW_ADMINISTRATOR = condor-admin@cs.wisc.edu/*.cs.wisc.edu

# condor daemon certificate(s) trusted by condor tools and daemons
# when connecting to other condor daemons
GSI_DAEMON_NAME = /C=US/O=Condor/O=UW/OU=CS/CN=condor@cs.wisc.edu

# clear out any host-based authorizations
# (unnecessary if you leave authentication REQUIRED,
#  but useful if you make it optional and want to
#  allow some unauthenticated operations, such as
#  ALLOW_READ = */*.cs.wisc.edu)
HOSTALLOW_READ =
HOSTALLOW_WRITE =
HOSTALLOW_NEGOTIATOR =
HOSTALLOW_ADMINISTRATOR =
```

The SEC_DEFAULT_AUTHENTICATION macro specifies that authentication is required for all communications. This single macro covers all communications, but could be replaced with a set of macros that require authentication for only specific communications.

The macro GSI_DAEMON_DIRECTORY is specified to give Condor a single place to find the daemon's certificate. This path may be a directory on a shared file system such as AFS.

Alternatively, this path name can point to local copies of the certificate stored in a local file system.

The macro GRIDMAP specifies the file to use for mapping GSI names to user names within Condor. For example, it might look like this:

```
"/C=US/O=Condor/O=UW/OU=CS/CN=condor@cs.wisc.edu" condor@cs.wisc.edu
```

Additional mappings would be needed for the users who submit jobs to the pool or who issue administrative commands.

SSL Authentication

SSL authentication is similar to GSI authentication, but without GSI's delegation (proxy) capabilities. SSL utilizes X.509 certificates.

All SSL authentication is mutual authentication in Condor. This means that when SSL authentication is used and when one process communicates with another, each process must be able to verify the signature on the certificate presented by the other process. The process that initiates the connection is the client, and the process that receives the connection is the server. For example, when a *condor_startd* daemon authenticates with a *condor_collector* daemon to provide a machine ClassAd, the *condor_startd* daemon initiates the connection and acts as the client, and the *condor_collector* daemon acts as the server.

The names and locations of keys and certificates for clients, servers, and the files used to specify trusted certificate authorities (CAs) are defined by settings in the configuration files. The contents of the files are identical in format and interpretation to those used by other systems which use SSL, such as Apache httpd.

The configuration variables AUTH_SSL_CLIENT_CERTFILE and AUTH_SSL_SERVER_CERTFILE specify the file location for the certificate file for the initiator and recipient of connections, respectively. Similarly, the configuration variables AUTH_SSL_CLIENT_KEYFILE and AUTH_SSL_SERVER_KEYFILE specify the locations for keys.

The configuration variables AUTH_SSL_SERVER_CAFILE and AUTH_SSL_CLIENT_CAFILE each specify a path and file name, providing the location of a file containing one or more certificates issued by trusted certificate authorities. Similarly, AUTH_SSL_SERVER_CADIR and AUTH_SSL_CLIENT_CADIR each specify a directory with one or more files, each which may contain a single CA certificate. The directories must be prepared using the OpenSSL c_rehash utility.

Kerberos Authentication

If Kerberos is used for authentication, then a mapping from a Kerberos domain (called a realm) to a Condor UID domain is necessary. There are two ways to accomplish this mapping. For a first

way to specify the mapping, see section 3.6.4 to use Condor's unified map file. A second way to specify the mapping defines the configuration variable KERBEROS_MAP_FILE to define a path to an administrator-maintained Kerberos-specific map file. The configuration syntax is

```
KERBEROS_MAP_FILE = /path/to/etc/condor.kmap
```

Lines within this map file have the syntax

```
KERB.REALM = UID.domain.name
```

Here are two lines from a map file to use as an example:

```
CS.WISC.EDU   = cs.wisc.edu
ENGR.WISC.EDU = ee.wisc.edu
```

If a KERBEROS_MAP_FILE configuration variable is defined and set, then all permitted realms must be explicitly mapped. If no map file is specified, then Condor assumes that the Kerberos realm is the same as the Condor UID domain.

The configuration variable KERBEROS_SERVER_PRINCIPAL defines the name of a Kerberos principal. If KERBEROS_SERVER_PRINCIPAL is not defined, then the default value used is host. A principal specifies a unique name to which a set of credentials may be assigned.

Condor takes the specified (or default) principal and appends a slash character, the host name, an '@' (at sign character), and the Kerberos realm. As an example, the configuration

```
KERBEROS_SERVER_PRINCIPAL = condor-daemon
```

results in Condor's use of

```
condor-daemon/the.host.name@YOUR.KERB.REALM
```

as the server principal.

Here is an example of configuration settings that use Kerberos for authentication and require authentication of all communications of the write or administrator access level.

```
SEC_WRITE_AUTHENTICATION                 = REQUIRED
SEC_WRITE_AUTHENTICATION_METHODS         = KERBEROS
SEC_ADMINISTRATOR_AUTHENTICATION         = REQUIRED
SEC_ADMINISTRATOR_AUTHENTICATION_METHODS = KERBEROS
```

Kerberos authentication on Unix platforms requires access to various files that usually are only accessible by the root user. At this time, the only supported way to use KERBEROS authentication on Unix platforms is to start daemons Condor as user root.

Password Authentication

The password method provides mutual authentication through the use of a shared secret. This is often a good choice when strong security is desired, but an existing Kerberos or X.509 infrastructure is not in place. Password authentication is available on both Unix and Windows. It currently can only be used for daemon-to-daemon authentication. The shared secret in this context is referred to as the *pool password*.

Before a daemon can use password authentication, the pool password must be stored on the daemon's local machine. On Unix, the password will be placed in a file defined by the configuration variable `SEC_PASSWORD_FILE`. This file will be accessible only by the UID that Condor is started as. On Windows, the same secure password store that is used for user passwords will be used for the pool password (see section 6.2.3).

Under Unix, the password file can be generated by using the following command to write directly to the password file:

```
condor_store_cred -f /path/to/password/file
```

Under Windows (or under Unix), storing the pool password is done with the **-c** option when using to *condor_store_cred* **add**. Running

```
condor_store_cred -c add
```

prompts for the pool password and store it on the local machine, making it available for daemons to use in authentication. The *condor_master* must be running for this command to work.

In addition, storing the pool password to a given machine requires `CONFIG`-level access. For example, if the pool password should only be set locally, and only by root, the following would be placed in the global configuration file.

```
ALLOW_CONFIG = root@mydomain/$(IP_ADDRESS)
```

It is also possible to set the pool password remotely, but this is recommended only if it can be done over an encrypted channel. This is possible on Windows, for example, in an environment where common accounts exist across all the machines in the pool. In this case, `ALLOW_CONFIG` can be set to allow the Condor administrator (who in this example has an account `condor` common to all machines in the pool) to set the password from the central manager as follows.

```
ALLOW_CONFIG = condor@mydomain/$(CONDOR_HOST)
```

The Condor administrator then executes

```
condor_store_cred -c -n host.mydomain add
```

from the central manager to store the password to a given machine. Since the `condor` account exists on both the central manager and `host.mydomain`, the NTSSPI authentication method can be used to authenticate and encrypt the connection. *condor_store_cred* will warn and prompt for cancellation, if the channel is not encrypted for whatever reason (typically because common accounts do not exist or Condor's security is misconfigured).

When a daemon is authenticated using a pool password, its security principle is `condor_pool@$(UID_DOMAIN)`, where `$(UID_DOMAIN)` is taken from the daemon's configuration. The `ALLOW_DAEMON` and `ALLOW_NEGOTIATOR` configuration variables for authorization should restrict access using this name. For example,

```
ALLOW_DAEMON = condor_pool@mydomain/*, condor@mydomain/$(IP_ADDRESS)
ALLOW_NEGOTIATOR = condor_pool@mydomain/$(CONDOR_HOST)
```

This configuration allows remote `DAEMON`-level and `NEGOTIATOR`-level access, if the pool password is known. Local daemons authenticated as `condor@mydomain` are also allowed access. This is done so local authentication can be done using another method such as `FS`.

Example Security Configuration Using Pool Password The following example configuration uses pool password authentication and network message integrity checking for all communication between Condor daemons.

```
SEC_PASSWORD_FILE = $(LOCK)/pool_password
SEC_DAEMON_AUTHENTICATION = REQUIRED
SEC_DAEMON_INTEGRITY = REQUIRED
SEC_DAEMON_AUTHENTICATION_METHODS = PASSWORD
SEC_NEGOTIATOR_AUTHENTICATION = REQUIRED
SEC_NEGOTIATOR_INTEGRITY = REQUIRED
SEC_NEGOTIATOR_AUTHENTICATION_METHODS = PASSWORD
SEC_CLIENT_AUTHENTICATION_METHODS = FS, PASSWORD, KERBEROS, GSI
ALLOW_DAEMON = condor_pool@$(UID_DOMAIN)/*.cs.wisc.edu, \
               condor@$(UID_DOMAIN)/$(IP_ADDRESS)
ALLOW_NEGOTIATOR = condor_pool@$(UID_DOMAIN)/negotiator.machine.name
```

Example Using Pool Password for *condor_startd* Advertisement One problem with the pool password method of authentication is that it involves a single, shared secret. This does not scale well with the addition of remote users who flock to the local pool. However, the pool password may still be used for authenticating portions of the local pool, while others (such as the remote *condor_schedd* daemons involved in flocking) are authenticated by other means.

In this example, only the *condor_startd* daemons in the local pool are required to have the pool password when they advertise themselves to the *condor_collector* daemon.

```
SEC_PASSWORD_FILE = $(LOCK)/pool_password
SEC_ADVERTISE_STARTD_AUTHENTICATION = REQUIRED
SEC_ADVERTISE_STARTD_INTEGRITY = REQUIRED
```

```
SEC_ADVERTISE_STARTD_AUTHENTICATION_METHODS = PASSWORD
SEC_CLIENT_AUTHENTICATION_METHODS = FS, PASSWORD, KERBEROS, GSI
ALLOW_ADVERTISE_STARTD = condor_pool@$(UID_DOMAIN)/*.cs.wisc.edu
```

File System Authentication

This form of authentication utilizes the ownership of a file in the identity verification of a client. A daemon authenticating a client requires the client to write a file in a specific location (`/tmp`). The daemon then checks the ownership of the file. The file's ownership verifies the identity of the client. In this way, the file system becomes the trusted authority. This authentication method is only appropriate for clients and daemons that are on the same computer.

File System Remote Authentication

Like file system authentication, this form of authentication utilizes the ownership of a file in the identity verification of a client. In this case, a daemon authenticating a client requires the client to write a file in a specific location, but the location is not restricted to `/tmp`. The location of the file is specified by the configuration variable `FS_REMOTE_DIR`.

Windows Authentication

This authentication is done only among Windows machines using a proprietary method. The Windows security interface SSPI is used to enforce NTLM (NT LAN Manager). The authentication is based on challenge and response, using the user's password as a key. This is similar to Kerberos. The main difference is that Kerberos provides an access token that typically grants access to an entire network, whereas NTLM authentication only verifies an identity to one machine at a time. NTSSPI is best-used in a way similar to file system authentication in Unix, and probably should not be used for authentication between two computers.

Claim To Be Authentication

Claim To Be authentication accepts any identity claimed by the client. As such, it does not authenticate. It is included in Condor and in the list of authentication methods for testing purposes only.

Anonymous Authentication

Anonymous authentication causes authentication to be skipped entirely. As such, it does not authenticate. It is included in Condor and in the list of authentication methods for testing purposes only.

3.6.4 The Unified Map File for Authentication

Condor's unified map file allows the mappings from authenticated names to a Condor canonical user name to be specified as a single list within a single file. The location of the unified map file is defined by the configuration variable `CERTIFICATE_MAPFILE`; it specifies the path and file name of the unified map file. Each mapping is on its own line of the unified map file. Each line contains 3 fields, separated by white space (space or tab characters):

1. The name of the authentication method to which the mapping applies.
2. A regular expression representing the authenticated name to be mapped.
3. The canonical Condor user name.

Allowable authentication method names are the same as used to define any of the configuration variables `SEC_*_AUTHENTICATION_METHODS`, as repeated here:

```
GSI
SSL
KERBEROS
PASSWORD
FS
FS_REMOTE
NTSSPI
CLAIMTOBE
ANONYMOUS
```

The fields that represent an authenticated name and the canonical Condor user name may utilize regular expressions as defined by PCRE (Perl-Compatible Regular Expressions). Due to this, more than one line (mapping) within the unified map file may match. Look ups are therefore defined to use the first mapping that matches.

A regular expression may need to contain spaces, and in this case the entire expression can be surrounded by double quote marks. If a double quote character also needs to appear in such an expression, it is preceded by a backslash.

The default behavior of Condor when no map file is specified is to do the following mappings, with some additional logic noted below:

```
FS (.*) \1
FS_REMOTE (.*) \1
GSI (.*) GSS_ASSIST_GRIDMAP
SSL (.*) ssl@unmapped
KERBEROS ([^/]*)/?[^@]*@(.*) \1@\2
NTSSPI (.*) \1
CLAIMTOBE (.*) \1
PASSWORD (.*) \1
```

For GSI (or SSL), the special name GSS_ASSIST_GRIDMAP instructs Condor to use the GSI grid map file (configured with GRIDMAP as shown in section 3.6.3) to do the mapping. If no mapping can be found for GSI (with or without the use of GSS_ASSIST_GRIDMAP), the user is mapped to gsi@unmapped.

For Kerberos, if KERBEROS_MAP_FILE is specified, the domain portion of the name is obtained by mapping the Kerberos realm to the value specified in the map file, rather than just using the realm verbatim as the domain portion of the condor user name. See section 3.6.3 for details.

If authentication did not happen or failed and was not required, then the user is given the name unauthenticated@unmapped.

With the integration of VOMS for GSI authentication, the interpretation of the regular expression representing the authenticated name may change. First, the full serialized DN and FQAN are used in attempting a match. If no match is found using the full DN and FQAN, then the DN is then used on its own without the FQAN. Using this, roles or user names from the VOMS attributes may be extracted to be used as the target for mapping. And, in this case the FQAN are verified, permitting reliance on their authenticity.

3.6.5 Encryption

Encryption provides privacy support between two communicating parties. Through configuration macros, both the client and the daemon can specify whether encryption is required for further communication.

The client uses one of two macros to enable or disable encryption:

```
SEC_DEFAULT_ENCRYPTION
SEC_CLIENT_ENCRYPTION
```

For the daemon, there are seven macros to enable or disable encryption:

```
SEC_DEFAULT_ENCRYPTION
SEC_READ_ENCRYPTION
SEC_WRITE_ENCRYPTION
SEC_ADMINISTRATOR_ENCRYPTION
SEC_CONFIG_ENCRYPTION
SEC_OWNER_ENCRYPTION
SEC_DAEMON_ENCRYPTION
SEC_NEGOTIATOR_ENCRYPTION
SEC_ADVERTISE_MASTER_ENCRYPTION
SEC_ADVERTISE_STARTD_ENCRYPTION
SEC_ADVERTISE_SCHEDD_ENCRYPTION
```

As an example, the macro defined in the configuration file for a daemon as

```
SEC_CONFIG_ENCRYPTION = REQUIRED
```

signifies that any communication that changes a daemon's configuration must be encrypted. If a daemon's configuration contains

```
SEC_DEFAULT_ENCRYPTION = REQUIRED
```

and does not contain any other security configuration for ENCRYPTION, then this default defines the daemon's needs for encryption over all access levels. Where a specific macro is present, its value takes precedence over any default given.

If encryption is to be done, then the communicating parties must find (negotiate) a mutually acceptable method of encryption to be used. A list of acceptable methods may be provided by the client, using the macros

```
SEC_DEFAULT_CRYPTO_METHODS
SEC_CLIENT_CRYPTO_METHODS
```

A list of acceptable methods may be provided by the daemon, using the macros

```
SEC_DEFAULT_CRYPTO_METHODS
SEC_READ_CRYPTO_METHODS
SEC_WRITE_CRYPTO_METHODS
SEC_ADMINISTRATOR_CRYPTO_METHODS
SEC_CONFIG_CRYPTO_METHODS
SEC_OWNER_CRYPTO_METHODS
SEC_DAEMON_CRYPTO_METHODS
SEC_NEGOTIATOR_CRYPTO_METHODS
SEC_ADVERTISE_MASTER_CRYPTO_METHODS
SEC_ADVERTISE_STARTD_CRYPTO_METHODS
SEC_ADVERTISE_SCHEDD_CRYPTO_METHODS
```

The methods are given as a comma-separated list of acceptable values. These variables list the encryption methods that are available to be used. The ordering of the list gives preference; the first item in the list indicates the highest preference. Possible values are

```
3DES
BLOWFISH
```

3.6.6 Integrity

An integrity check assures that the messages between communicating parties have not been tampered with. Any change, such as addition, modification, or deletion can be detected. Through configuration macros, both the client and the daemon can specify whether an integrity check is required of further communication.

The client uses one of two macros to enable or disable an integrity check:

```
SEC_DEFAULT_INTEGRITY
SEC_CLIENT_INTEGRITY
```

For the daemon, there are seven macros to enable or disable an integrity check:

```
SEC_DEFAULT_INTEGRITY
SEC_READ_INTEGRITY
SEC_WRITE_INTEGRITY
SEC_ADMINISTRATOR_INTEGRITY
SEC_CONFIG_INTEGRITY
SEC_OWNER_INTEGRITY
SEC_DAEMON_INTEGRITY
SEC_NEGOTIATOR_INTEGRITY
SEC_ADVERTISE_MASTER_INTEGRITY
SEC_ADVERTISE_STARTD_INTEGRITY
SEC_ADVERTISE_SCHEDD_INTEGRITY
```

As an example, the macro defined in the configuration file for a daemon as

```
SEC_CONFIG_INTEGRITY = REQUIRED
```

signifies that any communication that changes a daemon's configuration must have its integrity assured. If a daemon's configuration contains

```
SEC_DEFAULT_INTEGRITY = REQUIRED
```

and does not contain any other security configuration for `INTEGRITY`, then this default defines the daemon's needs for integrity checks over all access levels. Where a specific macro is present, its value takes precedence over any default given.

A signed MD5 check sum is currently the only available method for integrity checking. Its use is implied whenever integrity checks occur. If more methods are implemented, then there will be further macros to allow both the client and the daemon to specify which methods are acceptable.

3.6.7 Authorization

Authorization protects resource usage by granting or denying access requests made to the resources. It defines who is allowed to do what.

Authorization is defined in terms of users. An initial implementation provided authorization based on hosts (machines), while the current implementation relies on user-based authorization.

Section 3.6.9 on Setting Up IP/Host-Based Security in Condor describes the previous implementation. This IP/Host-Based security still exists, and it can be used, but significantly stronger and more flexible security can be achieved with the newer authorization based on fully qualified user names. This section discusses user-based authorization.

The authorization portion of the security of a Condor pool is based on a set of configuration macros. The macros list which user will be authorized to issue what request given a specific access level. When a daemon is to be authorized, its user name is the login under which the daemon is executed.

These configuration macros define a set of users that will be allowed to (or denied from) carrying out various Condor commands. Each access level may have its own list of authorized users. A complete list of the authorization macros:

```
ALLOW_READ
ALLOW_WRITE
ALLOW_ADMINISTRATOR
ALLOW_CONFIG
ALLOW_SOAP
ALLOW_OWNER
ALLOW_NEGOTIATOR
ALLOW_DAEMON
DENY_READ
DENY_WRITE
DENY_ADMINISTRATOR
DENY_SOAP
DENY_CONFIG
DENY_OWNER
DENY_NEGOTIATOR
DENY_DAEMON
```

In addition, the following are used to control authorization of specific types of Condor daemons when advertising themselves to the pool. If unspecified, these default to the broader `ALLOW_DAEMON` and `DENY_DAEMON` settings.

```
ALLOW_ADVERTISE_MASTER
ALLOW_ADVERTISE_STARTD
ALLOW_ADVERTISE_SCHEDD
DENY_ADVERTISE_MASTER
DENY_ADVERTISE_STARTD
DENY_ADVERTISE_SCHEDD
```

Each client side of a connection may also specify its own list of trusted servers. This is done using the following settings. Note that the FS and CLAIMTOBE authentication methods are not symmetric. The client is authenticated by the server, but the server is not authenticated by the client.

When the server is not authenticated to the client, only the network address of the host may be authorized and not the specific identity of the server.

```
ALLOW_CLIENT
DENY_CLIENT
```

The names `ALLOW_CLIENT` and `DENY_CLIENT` should be thought of as "when I am acting as a client, these are the servers I allow or deny." It should *not* be confused with the incorrect thought "when I am the server, these are the clients I allow or deny."

All authorization settings are defined by a comma-separated list of fully qualified users. Each fully qualified user is described using the following format:

```
username@domain/hostname
```

The information to the left of the slash character describes a user within a domain. The information to the right of the slash character describes one or more machines from which the user would be issuing a command. This host name may take the form of either a fully qualified host name of the form

```
bird.cs.wisc.edu
```

or an IP address of the form

```
128.105.128.0
```

An example is

```
zmiller@cs.wisc.edu/bird.cs.wisc.edu
```

Within the format, wild card characters (the asterisk, *) are allowed. The use of wild cards is limited to one wild card on either side of the slash character. A wild card character used in the host name is further limited to come at the beginning of a fully qualified host name or at the end of an IP address. For example,

```
*@cs.wisc.edu/bird.cs.wisc.edu
```

refers to any user that comes from `cs.wisc.edu`, where the command is originating from the machine `bird.cs.wisc.edu`. Another valid example,

```
zmiller@cs.wisc.edu/*.cs.wisc.edu
```

refers to commands coming from any machine within the `cs.wisc.edu` domain, and issued by `zmiller`. A third valid example,

```
*@cs.wisc.edu/*
```

refers to commands coming from any user within the `cs.wisc.edu` domain where the command is issued from any machine. A fourth valid example,

```
*@cs.wisc.edu/128.105.*
```

refers to commands coming from any user within the `cs.wisc.edu` domain where the command is issued from machines within the network that match the first two octets of the IP address.

If the set of machines is specified by an IP address, then further specification using a net mask identifies a physical set (subnet) of machines. This physical set of machines is specified using the form

```
network/netmask
```

The `network` is an IP address. The net mask takes one of two forms. It may be a decimal number which refers to the number of leading bits of the IP address that are used in describing a subnet. Or, the net mask may take the form of

```
a.b.c.d
```

where `a`, `b`, `c`, and `d` are decimal numbers that each specify an 8-bit mask. An example net mask is

```
255.255.192.0
```

which specifies the bit mask

```
11111111.11111111.11000000.00000000
```

A single complete example of a configuration variable that uses a net mask is

```
ALLOW_WRITE = joesmith@cs.wisc.edu/128.105.128.0/17
```

User `joesmith` within the `cs.wisc.edu` domain is given write authorization when originating from machines that match their leftmost 17 bits of the IP address.

This flexible set of configuration macros could used to define conflicting authorization. Therefore, the following protocol defines the precedence of the configuration macros.

1. DENY_* macros take precedence over ALLOW_* macros where there is a conflict. This implies that if a specific user is both denied and granted authorization, the conflict is resolved by denying access.

2. If macros are omitted, the default behavior is to grant authorization for every user.

In addition, there are some hard-coded authorization rules that cannot be modified by configuration.

1. Connections with a name matching *@unmapped are not allowed to do any job management commands (e.g. submitting, removing, or modifying jobs). This prevents these operations from being done by unauthenticated users and users who are authenticated but lacking a name in the map file.

2. To simplify flocking, the *condor_schedd* automatically grants the *condor_startd* READ access for the duration of a claim so that claim-related communications are possible. The *condor_shadow* grants the *condor_starter* DAEMON access so that file transfers can be done. The identity that is granted access in both these cases is the authenticated name (if available) and IP address of the *condor_startd* when the *condor_schedd* initially connects to it to request the claim. It is important that only trusted *condor_startd*s are allowed to publish themselves to the collector or that the *condor_schedd*'s ALLOW_CLIENT setting prevent it from allowing connections to *condor_startd*s that it does not trust to run jobs.

3. When SEC_ENABLE_MATCH_PASSWORD_AUTHENTICATION is true, execute-side@matchsession is automatically granted READ access to the *condor_schedd* and DAEMON access to the *condor_shadow*.

Example of Authorization Security Configuration

An example of the configuration variables for the user-side authorization is derived from the necessary access levels as described in Section 3.6.1.

```
ALLOW_READ            = *@cs.wisc.edu/*
ALLOW_WRITE           = *@cs.wisc.edu/*.cs.wisc.edu
ALLOW_ADMINISTRATOR   = condor-admin@cs.wisc.edu/*.cs.wisc.edu
ALLOW_CONFIG          = condor-admin@cs.wisc.edu/*.cs.wisc.edu
ALLOW_NEGOTIATOR      = condor@cs.wisc.edu/condor.cs.wisc.edu, \
                        condor@cs.wisc.edu/condor2.cs.wisc.edu
ALLOW_DAEMON          = condor@cs.wisc.edu/*.cs.wisc.edu

# Clear out any old-style HOSTALLOW settings:
HOSTALLOW_READ =
HOSTALLOW_WRITE =
HOSTALLOW_DAEMON =
HOSTALLOW_NEGOTIATOR =
HOSTALLOW_ADMINISTRATOR =
HOSTALLOW_OWNER =
```

This example configuration authorizes any authenticated user in the `cs.wisc.edu` domain to carry out a request that requires the `READ` access level from any machine. Any user in the `cs.wisc.edu` domain may carry out a request that requires the `WRITE` access level from any machine in the `cs.wisc.edu` domain. Only the user called `condor-admin` may carry out a request that requires the `ADMINISTRATOR` access level from any machine in the `cs.wisc.edu` domain. The administrator, logged into any machine within the `cs.wisc.edu` domain is authorized at the `CONFIG` access level. Only the negotiator daemon, running as `condor` on the two central managers are authorized with the `NEGOTIATOR` access level. And, the last line of the example presumes that there is a user called condor, and that the daemons have all been started up as this user. It authorizes only programs (which will be the daemons) running as `condor` to carry out requests that require the `DAEMON` access level, where the commands originate from any machine in the `cs.wisc.edu` domain.

In the local configuration file for each host, the host's owner should be authorized as the owner of the machine. An example of the entry in the local configuration file:

```
ALLOW_OWNER           = username@cs.wisc.edu/hostname.cs.wisc.edu
```

In this example the owner has a login of `username`, and the machine's name is represented by `hostname`.

Debugging Security Configuration

If the authorization policy denies a network request, an explanation of why the request was denied is printed in the log file of the daemon that denied the request. The line in the log file contains the words `PERMISSION DENIED`.

To get Condor to generate a similar explanation of why requests are accepted, add `D_SECURITY` to the daemon's debug options (and restart or reconfig the daemon). The line in the log file for these cases will contain the words `PERMISSION GRANTED`. If you do not want to see a full explanation but just want to see when requests are made, add `D_COMMAND` to the daemon's debug options.

If the authorization policy makes use of host or domain names, then be aware that Condor depends on DNS to map IP addresses to names. The security and accuracy of your DNS service is therefore a requirement. Typos in DNS mappings are an occasional source of unexpected behavior. If the authorization policy is not behaving as expected, carefully compare the names in the policy with the host names Condor mentions in the explanations of why requests are granted or denied.

3.6.8 Security Sessions

To set up and configure secure communications in Condor, authentication, encryption, and integrity checks can be used. However, these come at a cost: performing strong authentication can take a significant amount of time, and generating the cryptographic keys for encryption and integrity checks can take a significant amount of processing power.

The Condor system makes many network connections between different daemons. If each one of these was to be authenticated, and new keys were generated for each connection, Condor would not be able to scale well. Therefore, Condor uses the concept of *sessions* to cache relevant security information for future use and greatly speed up the establishment of secure communications between the various Condor daemons.

A new session is established the first time a connection is made from one daemon to another. Each session has a fixed lifetime after which it will expire and a new session will need to be created again. But while a valid session exists, it can be re-used as many times as needed, thereby preventing the need to continuously re-establish secure connections. Each entity of a connection will have access to a *session key* that proves the identity of the other entity on the opposing side of the connection. This session key is exchanged securely using a strong authentication method, such as Kerberos or GSI. Other authentication methods, such as `NTSSPI`, `FS_REMOTE`, `CLAIMTOBE`, and `ANONYMOUS`, do not support secure key exchange. An entity listening on the wire may be able to impersonate the client or server in a session that does not use a strong authentication method.

Establishing a secure session requires that either the encryption or the integrity options be enabled. If the encryption capability is enabled, then the session will be restarted using the session key as the encryption key. If integrity capability is enabled, then the check sum includes the session key even though it is not transmitted. Without either of these two methods enabled, it is possible for an attacker to use an open session to make a connection to a daemon and use that connection for nefarious purposes. It is strongly recommended that if *you have authentication turned on, you should also turn on integrity and/or encryption.*

The configuration parameter `SEC_DEFAULT_NEGOTIATION` will allow a user to set the default level of secure sessions in Condor. Like other security settings, the possible values for this parameter can be `REQUIRED`, `PREFERRED`, `OPTIONAL`, or `NEVER`. If you disable sessions and you have authentication turned on, then most authentication (other than commands like *condor_submit*) will fail because Condor requires sessions when you have security turned on. On the other hand, if you are not using strong security in Condor, but you are relying on the default host-based security, turning off sessions may be useful in certain situations. These might include debugging problems with the security session management or slightly decreasing the memory consumption of the daemons, which keep track of the sessions in use.

Session lifetimes for specific daemons are already properly configured in the default installation of Condor. Condor tools such as *condor_q* and *condor_status* create a session that expires after one minute. Theoretically they should not create a session at all, because the session cannot be reused between program invocations, but this is difficult to do in the general case. This allows a very small window of time for any possible attack, and it helps keep the memory footprint of running daemons down, because they are not keeping track of all of the sessions. The session durations may be manually tuned by using macros in the configuration file, but this is not recommended.

3.6.9 Host-Based Security in Condor

This section describes the mechanisms for setting up Condor's host-based security. This is now an outdated form of implementing security levels for machine access. It remains available and

documented for purposes of backward compatibility. If used at the same time as the user-based authorization, the two specifications are merged together.

The host-based security paradigm allows control over which machines can join a Condor pool, which machines can find out information about your pool, and which machines within a pool can perform administrative commands. By default, Condor is configured to allow anyone to view or join a pool. It is recommended that this parameter is changed to only allow access from machines that you trust.

This section discusses how the host-based security works inside Condor. It lists the different levels of access and what parts of Condor use which levels. There is a description of how to configure a pool to grant or deny certain levels of access to various machines. Configuration examples and the settings of configuration variables using the *condor_config_val* command complete this section.

Inside the Condor daemons or tools that use DaemonCore (see section 3.9 for details), most tasks are accomplished by sending commands to another Condor daemon. These commands are represented by an integer value to specify which command is being requested, followed by any optional information that the protocol requires at that point (such as a ClassAd, capability string, etc). When the daemons start up, they will register which commands they are willing to accept, what to do with arriving commands, and the access level required for each command. When a command request is received by a daemon, Condor identifies the access level required and checks the IP address of the sender to verify that it satisfies the allow/deny settings from the configuration file. If permission is granted, the command request is honored; otherwise, the request will be aborted.

Settings for the access levels in the global configuration file will affect all the machines in the pool. Settings in a local configuration file will only affect the specific machine. The settings for a given machine determine what other hosts can send commands to that machine. If a machine foo is to be given administrator access on machine bar, place foo in bar's configuration file access list (not the other way around).

The following are the various access levels that commands within Condor can be registered with:

READ Machines with `READ` access can read information from the Condor daemons. For example, they can view the status of the pool, see the job queue(s), and view user permissions. `READ` access does not allow a machine to alter any information, and does not allow job submission. A machine listed with `READ` permission will be unable join a Condor pool; the machine can only view information about the pool.

WRITE Machines with `WRITE` access can write information to the Condor daemons. Most important for granting a machine with this access is that the machine will be able to join a pool since they are allowed to send ClassAd updates to the central manager. The machine can talk to the other machines in a pool in order to submit or run jobs. In addition, any machine with `WRITE` access can request the *condor_startd* daemon to perform periodic checkpoints on an executing job. After the checkpoint is completed, the job will continue to execute and the machine will still be claimed by the original *condor_schedd* daemon. This allows users on the machines where they submitted their jobs to use the *condor_checkpoint* command to get their jobs to periodically checkpoint, even if the users do not have an account on the machine where the jobs execute.

IMPORTANT: For a machine to join a Condor pool, the machine must have both `WRITE` permission **AND** `READ` permission. `WRITE` permission is not enough.

`ADMINISTRATOR` Machines with `ADMINISTRATOR` access are granted additional Condor administrator rights to the pool. This includes the ability to change user priorities (with the command `userprio -set`), and the ability to turn Condor on and off (with the command *`condor_off`* <`machine`>). It is recommended that few machines be granted administrator access in a pool; typically these are the machines that are used by Condor and system administrators as their primary workstations, or the machines running as the pool's central manager.

IMPORTANT: Giving `ADMINISTRATOR` privileges to a machine grants administrator access for the pool to **ANY USER** on that machine. This includes any users who can run Condor jobs on that machine. It is recommended that `ADMINISTRATOR` access is granted with due diligence.

`OWNER` This level of access is required for commands that the owner of a machine (any local user) should be able to use, in addition to the Condor administrators. For example, the *condor_vacate* command causes the *condor_startd* daemon to vacate any running Condor job. It requires `OWNER` permission, so that any user logged into a local machine can issue a *condor_vacate* command.

`NEGOTIATOR` This access level is used specifically to verify that commands are sent by the *condor_negotiator* daemon. The *condor_negotiator* daemon runs on the central manager of the pool. Commands requiring this access level are the ones that tell the *condor_schedd* daemon to begin negotiating, and those that tell an available *condor_startd* daemon that it has been matched to a *condor_schedd* with jobs to run.

`CONFIG` This access level is required to modify a daemon's configuration using the *condor_config_val* command. By default, machines with this level of access are able to change any configuration parameter, except those specified in the `condor_config.root` configuration file. Therefore, one should exercise extreme caution before granting this level of host-wide access. Because of the implications caused by `CONFIG` privileges, it is disabled by default for all hosts.

`DAEMON` This access level is used for commands that are internal to the operation of Condor. An example of this internal operation is when the *condor_startd* daemon sends its ClassAd updates to the *condor_collector* daemon (which may be more specifically controlled by the `ADVERTISE_STARTD` access level). Authorization at this access level should only be given to hosts that actually run Condor in your pool. The `DAEMON` level of access implies both `READ` and `WRITE` access. Any setting for this access level that is not defined will default to the corresponding setting in the `WRITE` access level.

`ADVERTISE_MASTER` This access level is used specifically for commands used to advertise a *condor_master* daemon to the collector. Any setting for this access level that is not defined will default to the corresponding setting in the `DAEMON` access level.

`ADVERTISE_STARTD` This access level is used specifically for commands used to advertise a *condor_startd* daemon to the collector. Any setting for this access level that is not defined will default to the corresponding setting in the `DAEMON` access level.

ADVERTISE_SCHEDD This access level is used specifically for commands used to advertise a *condor_schedd* daemon to the collector. Any setting for this access level that is not defined will default to the corresponding setting in the DAEMON access level.

CLIENT This access level is different from all the others. Whereas all of the other access levels refer to the security policy for accepting connections *from* others, the CLIENT access level applies when a Condor daemon or tool is connecting *to* some other Condor daemon. In other words, it specifies the policy of the client that is initiating the operation, rather than the server that is being contacted.

ADMINISTRATOR and NEGOTIATOR access default to the central manager machine. OWNER access defaults to the local machine, as well as any machines given with ADMINISTRATOR access. CONFIG access is not granted to any machine as its default. These defaults are sufficient for most pools, and should not be changed without a compelling reason. If machines other than the default are to have to have OWNER access, they probably should also have ADMINISTRATOR access. By granting machines ADMINISTRATOR access, they will automatically have OWNER access, given how OWNER access is set within the configuration.

3.6.10 Examples of Security Configuration

Here is a sample security configuration:

```
ALLOW_ADMINISTRATOR = $(CONDOR_HOST)
ALLOW_OWNER = $(FULL_HOSTNAME), $(ALLOW_ADMINISTRATOR)
ALLOW_READ = *
ALLOW_WRITE = *
ALLOW_NEGOTIATOR = $(COLLECTOR_HOST)
ALLOW_NEGOTIATOR_SCHEDD = $(COLLECTOR_HOST), $(FLOCK_NEGOTIATOR_HOSTS)
ALLOW_WRITE_COLLECTOR = $(ALLOW_WRITE), $(FLOCK_FROM)
ALLOW_WRITE_STARTD    = $(ALLOW_WRITE), $(FLOCK_FROM)
ALLOW_READ_COLLECTOR  = $(ALLOW_READ), $(FLOCK_FROM)
ALLOW_READ_STARTD     = $(ALLOW_READ), $(FLOCK_FROM)
ALLOW_CLIENT = *
```

This example configuration presumes that the *condor_collector* and *condor_negotiator* daemons are running on the same machine.

For each access level, an ALLOW or a DENY may be added.

- If there is an ALLOW, it means "only allow these machines". No ALLOW means allow anyone.
- If there is a DENY, it means "deny these machines". No DENY means deny nobody.
- If there is both an ALLOW and a DENY, it means allow the machines listed in ALLOW except for the machines listed in DENY.

- Exclusively for the CONFIG access, no ALLOW means allow no one. Note that this is different than the other ALLOW configurations. It is different to enable more stringent security where older configurations are used, since older configuration files would not have a CONFIG configuration entry.

Multiple machine entries in the configuration files may be separated by either a space or a comma. The machines may be listed by

- Individual host names, for example: condor.cs.wisc.edu
- Individual IP address, for example: 128.105.67.29
- IP subnets (use a trailing *), for example: 144.105.*, 128.105.67.*
- Host names with a wild card * character (only one * is allowed per name), for example: *.cs.wisc.edu, sol*.cs.wisc.edu

To resolve an entry that falls into both allow and deny: individual machines have a higher order of precedence than wild card entries, and host names with a wild card have a higher order of precedence than IP subnets. Otherwise, DENY has a higher order of precedence than ALLOW. This is how most people would intuitively expect it to work.

In addition, the above access levels may be specified on a per-daemon basis, instead of machine-wide for all daemons. Do this with the subsystem string (described in section 3.3.1 on Subsystem Names), which is one of: STARTD, SCHEDD, MASTER, NEGOTIATOR, or COLLECTOR. For example, to grant different read access for the *condor_schedd*:

```
ALLOW_READ_SCHEDD = <list of machines>
```

Here are more examples of configuration settings. Notice that ADMINISTRATOR access is only granted through an ALLOW setting to explicitly grant access to a small number of machines. We recommend this.

- Let any machine join the pool. Only the central manager has administrative access.

  ```
  ALLOW_ADMINISTRATOR = $(CONDOR_HOST)
  ALLOW_OWNER = $(FULL_HOSTNAME), $(ALLOW_ADMINISTRATOR)
  ```

- Only allow machines at NCSA to join or view the pool. The central manager is the only machine with ADMINISTRATOR access.

  ```
  ALLOW_READ = *.ncsa.uiuc.edu
  ALLOW_WRITE = *.ncsa.uiuc.edu
  ALLOW_ADMINISTRATOR = $(CONDOR_HOST)
  ALLOW_OWNER = $(FULL_HOSTNAME), $(ALLOW_ADMINISTRATOR)
  ```

- Only allow machines at NCSA and the U of I Math department join the pool, *except do not* allow lab machines to do so. Also, do not allow the 177.55 subnet (perhaps this is the dial-in subnet). Allow anyone to view pool statistics. The machine named bigcheese administers the pool (not the central manager).

```
ALLOW_WRITE = *.ncsa.uiuc.edu, *.math.uiuc.edu
DENY_WRITE = lab-*.edu, *.lab.uiuc.edu, 177.55.*
ALLOW_ADMINISTRATOR = bigcheese.ncsa.uiuc.edu
ALLOW_OWNER = $(FULL_HOSTNAME), $(ALLOW_ADMINISTRATOR)
```

- Only allow machines at NCSA and UW-Madison's CS department to view the pool. Only NCSA machines and the machine raven.cs.wisc.edu can join the pool. Note: the machine raven.cs.wisc.edu has the read access it needs through the wild card setting in ALLOW_READ). This example also shows how to use the continuation character, \, to continue a long list of machines onto multiple lines, making it more readable. This works for all configuration file entries, not just host access entries.

```
ALLOW_READ = *.ncsa.uiuc.edu, *.cs.wisc.edu
ALLOW_WRITE = *.ncsa.uiuc.edu, raven.cs.wisc.edu
ALLOW_ADMINISTRATOR = $(CONDOR_HOST), bigcheese.ncsa.uiuc.edu, \
                      biggercheese.uiuc.edu
ALLOW_OWNER = $(FULL_HOSTNAME), $(ALLOW_ADMINISTRATOR)
```

- Allow anyone except the military to view the status of the pool, but only let machines at NCSA view the job queues. Only NCSA machines can join the pool. The central manager, bigcheese, and biggercheese can perform most administrative functions. However, only biggercheese can update user priorities.

```
DENY_READ = *.mil
ALLOW_READ_SCHEDD = *.ncsa.uiuc.edu
ALLOW_WRITE = *.ncsa.uiuc.edu
ALLOW_ADMINISTRATOR = $(CONDOR_HOST), bigcheese.ncsa.uiuc.edu, \
                      biggercheese.uiuc.edu
ALLOW_ADMINISTRATOR_NEGOTIATOR = biggercheese.uiuc.edu
ALLOW_OWNER = $(FULL_HOSTNAME), $(ALLOW_ADMINISTRATOR)
```

3.6.11 Changing the Security Configuration

A new security feature introduced in Condor version 6.3.2 enables more fine-grained control over the configuration settings that can be modified remotely with the *condor_config_val* command. The manual page for *condor_config_val* on page 718 details how to use *condor_config_val* to modify configuration settings remotely. Since certain configuration attributes can have a large impact on the functioning of the Condor system and the security of the machines in a Condor pool, it is important to restrict the ability to change attributes remotely.

For each security access level described, the Condor administrator can define which configuration settings a host at that access level is allowed to change. Optionally, the administrator can define separate lists of settable attributes for each Condor daemon, or the administrator can define one list that is used by all daemons.

For each command that requests a change in configuration setting, Condor searches all the different possible security access levels to see which, if any, the request satisfies. (Some hosts can qualify for multiple access levels. For example, any host with ADMINISTRATOR permission probably has WRITE permission also). Within the qualified access level, Condor searches for the list of attributes that may be modified. If the request is covered by the list, the request will be granted. If not covered, the request will be refused.

The default configuration shipped with Condor is exceedingly restrictive. Condor users or administrators cannot set configuration values from remote hosts with *condor_config_val*. Enabling this feature requires a change to the settings in the configuration file. Use this security feature carefully. Grant access only for attributes which you need to be able to modify in this manner, and grant access only at the most restrictive security level possible.

The most secure use of this feature allows Condor users to set attributes in the configuration file which are not used by Condor directly. These are custom attributes published by various Condor daemons with the <SUBSYS>_ATTRS setting described in section 3.3.5 on page 177. It is secure to grant access only to modify attributes that are used by Condor to publish information. Granting access to modify settings used to control the behavior of Condor is not secure. The goal is to ensure no one can use the power to change configuration attributes to compromise the security of your Condor pool.

The control lists are defined by configuration settings that contain SETTABLE_ATTRS in their name. The name of the control lists have the following form:

```
<SUBSYS>.SETTABLE_ATTRS_<PERMISSION-LEVEL>
```

The two parts of this name that can vary are the <PERMISSION-LEVEL> and the <SUBSYS>. The <PERMISSION-LEVEL> can be any of the security access levels described earlier in this section. Examples include WRITE, OWNER, and CONFIG.

The <SUBSYS> is an optional portion of the name. It can be used to define separate rules for which configuration attributes can be set for each kind of Condor daemon (for example, STARTD, SCHEDD, and MASTER). There are many configuration settings that can be defined differently for each daemon that use this <SUBSYS> naming convention. See section 3.3.1 on page 159 for a list. If there is no daemon-specific value for a given daemon, Condor will look for SETTABLE_ATTRS_<PERMISSION-LEVEL>.

Each control list is defined by a comma-separated list of attribute names which should be allowed to be modified. The lists can contain wild cards characters (*).

Some examples of valid definitions of control lists with explanations:

- SETTABLE_ATTRS_CONFIG = *

  Grant unlimited access to modify configuration attributes to any request that came from a machine in the CONFIG access level. This was the default behavior before Condor version 6.3.2.

- SETTABLE_ATTRS_ADMINISTRATOR = *_DEBUG, MAX_*_LOG

  Grant access to change any configuration setting that ended with _DEBUG (for example, STARTD_DEBUG) and any attribute that matched MAX_*_LOG (for example, MAX_SCHEDD_LOG) to any host with ADMINISTRATOR access.

- STARTD.SETTABLE_ATTRS_OWNER = HasDataSet

Allows any request to modify the HasDataSet attribute that came from a host with OWNER access. By default, OWNER covers any request originating from the local host, plus any machines listed in the ADMINISTRATOR level. Therefore, any Condor job would qualify for OWNER access to the machine where it is running. So, this setting would allow any process running on a given host, including a Condor job, to modify the HasDataSet variable for that host. HasDataSet is not used by Condor, it is an invented attribute included in the STARTD_ATTRS setting in order for this example to make sense.

3.6.12 Using Condor w/ Firewalls, Private Networks, and NATs

This topic is now addressed in more detail in section 3.7, which explains network communication in Condor.

3.6.13 User Accounts in Condor on Unix Platforms

On a Unix system, UIDs (User IDentification numbers) form part of an operating system's tools for maintaining access control. Each executing program has a UID, a unique identifier of a user executing the program. This is also called the real UID. A common situation has one user executing the program owned by another user. Many system commands work this way, with a user (corresponding to a person) executing a program belonging to (owned by) root. Since the program may require privileges that root has which the user does not have, a special bit in the program's protection specification (a setuid bit) allows the program to run with the UID of the program's owner, instead of the user that executes the program. This UID of the program's owner is called an effective UID.

Condor works most smoothly when its daemons run as root. The daemons then have the ability to switch their effective UIDs at will. When the daemons run as root, they normally leave their effective UID and GID (Group IDentification) to be those of user and group condor. This allows access to the log files without changing the ownership of the log files. It also allows access to these files when the user condor's home directory resides on an NFS server. root can not normally access NFS files.

If there is no condor user and group on the system, an administrator can specify which UID and GID the Condor daemons should use when they do not need root privileges in two ways: either with the CONDOR_IDS environment variable or the CONDOR_IDS configuration variable. In either case, the value should be the UID integer, followed by a period, followed by the GID integer. For example, if a Condor administrator does not want to create a condor user, and instead wants their Condor daemons to run as the daemon user (a common non-root user for system daemons to execute as), the daemon user's UID was 2, and group daemon had a GID of 2, the corresponding setting in the Condor configuration file would be CONDOR_IDS = 2.2.

On a machine where a job is submitted, the *condor_schedd* daemon changes its effective UID to root such that it has the capability to start up a *condor_shadow* daemon for the job. Before a *condor_shadow* daemon is created, the *condor_schedd* daemon switches back to root, so that it can start up the *condor_shadow* daemon with the (real) UID of the user who submitted the job. Since

the *condor_shadow* runs as the owner of the job, all remote system calls are performed under the owner's UID and GID. This ensures that as the job executes, it can access only files that its owner could access if the job were running locally, without Condor.

On the machine where the job executes, the job runs either as the submitting user or as user `nobody`, to help ensure that the job cannot access local resources or do harm. If the `UID_DOMAIN` matches, and the user exists as the same UID in password files on both the submitting machine and on the execute machine, the job will run as the submitting user. If the user does not exist in the execute machine's password file and `SOFT_UID_DOMAIN` is True, then the job will run under the submitting user's UID anyway (as defined in the submitting machine's password file). If `SOFT_UID_DOMAIN` is False, and `UID_DOMAIN` matches, and the user is not in the execute machine's password file, then the job execution attempt will be aborted.

Running Condor as Non-Root

While we strongly recommend starting up the Condor daemons as `root`, we understand that it is not always possible to do so. The main problems of not running Condor daemons as `root` appear when one Condor installation is shared by many users on a single machine, or if machines are set up to only execute Condor jobs. With a submit-only installation for a single user, there is no need for or benefit from running as `root`.

The effects of Condor of running both with and without root access are classified for each daemon:

condor_startd A Condor machine set up to execute jobs where the *condor_startd* is not started as `root` relies on the good will of the Condor users to agree to the policy configured for the *condor_startd* to enforce for starting, suspending, vacating, and killing Condor jobs. When the *condor_startd* is started as `root`, however, these policies may be enforced regardless of malicious users. By running as `root`, the Condor daemons run with a different UID than the Condor job. The user's job is started as either the UID of the user who submitted it, or as user `nobody`, depending on the `UID_DOMAIN` settings. Therefore, the Condor job cannot do anything to the Condor daemons. Without starting the daemons as `root`, all processes started by Condor, including the user's job, run with the same UID. Only `root` can switch UIDs. Therefore, a user's job could kill the *condor_startd* and *condor_starter*. By doing so, the user's job avoids getting suspended or vacated. This is nice for the job, as it obtains unlimited access to the machine, but it is awful for the machine owner or administrator. If there is trust of the users submitting jobs to Condor, this might not be a concern. However, to ensure that the policy chosen is enforced by Condor, the *condor_startd* should be started as `root`.

In addition, some system information cannot be obtained without `root` access on some platforms. As a result, when running without `root` access, the *condor_startd* must call other programs such as *uptime*, to get this information. This is much less efficient than getting the information directly from the kernel, as is done when running as `root`. On Linux, this information is available without root access, so it is not a concern on those platforms.

If all of Condor cannot be run as `root`, at least consider installing the *condor_startd* as setuid root. That would solve both problems. Barring that, install it as a setgid sys or kmem

program, depending on whatever group has read access to `/dev/kmem` on the system. That would solve the system information problem.

condor_schedd The biggest problem with running the *condor_schedd* without `root` access is that the *condor_shadow* processes which it spawns are stuck with the same UID that the *condor_schedd* has. This requires users to go out of their way to grant write access to user or group that the *condor_schedd* is run as for any files or directories their jobs write or create. Similarly, read access must be granted to their input files.

Consider installing *condor_submit* as a setgid `condor` program so that at least the `stdout`, `stderr` and user log files get created with the right permissions. If *condor_submit* is a setgid program, it will automatically set its umask to 002 and create group-writable files. This way, the simple case of a job that only writes to `stdout` and `stderr` will work. If users have programs that open their own files, they will need to know and set the proper permissions on the directories they submit from.

condor_master The *condor_master* spawns both the *condor_startd* and the *condor_schedd*. To have both running as `root`, have the *condor_master* run as `root`. This happens automatically if the *condor_master* is started from boot scripts.

condor_negotiator* and *condor_collector There is no need to have either of these daemons running as `root`.

condor_kbdd On platforms that need the *condor_kbdd*, the *condor_kbdd* must run as `root`. If it is started as any other user, it will not work. Consider installing this program as a setuid root binary if the *condor_master* will not be run as `root`. Without the *condor_kbdd*, the *condor_startd* has no way to monitor USB mouse or keyboard activity, although it will notice keyboard activity on ttys such as xterms and remote logins.

If Condor is not run as root, then choose almost any user name. A common choice is to set up and use the `condor` user; this simplifies the setup, because Condor will look for its configuration files in the `condor` user's directory. If `condor` is not selected, then the configuration must be placed properly such that Condor can find its configuration files.

If users will be submitting jobs as a user different than the user Condor is running as (perhaps you are running as the `condor` user and users are submitting as themselves), then users have to be careful to only have file permissions properly set up to be accessible by the user Condor is using. In practice, this means creating world-writable directories for output from Condor jobs. This creates a potential security risk, in that any user on the machine where the job is submitted can alter the data, remove it, or do other undesirable things. It is only acceptable in an environment where users can trust other users.

Normally, users without `root` access who wish to use Condor on their machines create a `condor` home directory somewhere within their own accounts and start up the daemons (to run with the UID of the user). As in the case where the daemons run as user `condor`, there is no ability to switch UIDs or GIDs. The daemons run as the UID and GID of the user who started them. On a machine where jobs are submitted, the *condor_shadow* daemons all run as this same user. But, if other users are using Condor on the machine in this environment, the *condor_shadow* daemons for

these other users' jobs execute with the UID of the user who started the daemons. This is a security risk, since the Condor job of the other user has access to all the files and directories of the user who started the daemons. Some installations have this level of trust, but others do not. Where this level of trust does not exist, it is best to set up a `condor` account and group, or to have each user start up their own Personal Condor submit installation.

When a machine is an execution site for a Condor job, the Condor job executes with the UID of the user who started the *condor_startd* daemon. This is also potentially a security risk, which is why we do not recommend starting up the execution site daemons as a regular user. Use either `root` or a user such as `condor` that exists only to run Condor jobs.

Running Jobs as the Nobody User

Under Unix, Condor runs jobs either as the user that submitted the jobs, or as the user called `nobody`. Condor uses user `nobody` if the value of the `UID_DOMAIN` configuration variable of the submitting and executing machines are different or if `STARTER_ALLOW_RUNAS_OWNER` is false or if the job ClassAd contains `RunAsOwner`=False. Under Windows, Condor by default runs jobs under a dynamically created local account that exists for the duration of the job, but it can optionally run the job as the user account that owns the job if `STARTER_ALLOW_RUNAS_OWNER` is True and the job contains `RunAsOwner`=True.

When Condor cleans up after executing a vanilla universe job, it does the best that it can by deleting all of the processes started by the job. During the life of the job, it also does its best to track the CPU usage of all processes created by the job. There are a variety of mechanisms used by Condor to detect all such processes, but, in general, the only foolproof mechanism is for the job to run under a dedicated execution account (as it does under Windows by default). With all other mechanisms, it is possible to fool Condor, and leave processes behind after Condor has cleaned up. In the case of a shared account, such as the Unix user `nobody`, it is possible for the job to leave a lurker process lying in wait for the next job run as `nobody`. The lurker process may prey maliciously on the next `nobody` user job, wreaking havoc.

Condor could prevent this problem by simply killing all processes run by the `nobody` user, but this would annoy many system administrators. The `nobody` user is often used for non-Condor system processes. It may also be used by other Condor jobs running on the same machine, if it is a multi-processor machine.

Condor provides a two-part solution to this difficulty. First, create user accounts specifically for Condor to use instead of user `nobody`. These can be low-privilege accounts, as the `nobody` user is. Create one of these accounts for each job execution slot per computer, so that distinct users can be used for concurrent processes. This prevents malicious behavior between processes running on distinct slots. Section 3.13.9 details slots. For a sample machine with two compute slots, create two users that are intended only to be used by Condor. As an example, call them `cndrusr1` and `cndrusr2`. Tell Condor about these users with the `SLOT<N>_USER` configuration variables, where `<N>` is replaced with the slot number. In this example:

```
SLOT1_USER = cndrusr1
```

```
SLOT2_USER = cndrusr2
```

Then tell Condor that these accounts are intended only to be used by Condor, so Condor can kill all the processes belonging to these users upon job completion. The configuration variable DEDICATED_EXECUTE_ACCOUNT_REGEXP is introduced and set to a regular expression that matches the account names we have just created.

```
DEDICATED_EXECUTE_ACCOUNT_REGEXP = cndrusr[0-9]+
```

Finally, tell Condor not to run jobs as the job owner:

```
STARTER_ALLOW_RUNAS_OWNER = False
```

Notes:

1. Currently, none of these configuration settings apply to standard universe jobs. Normally, standard universe jobs do not create additional processes.
2. On Windows, SLOT<N>_USER will only work if the credential of the specified user is stored on the execute machine using *condor_store_cred*. See the *condor_store_cred* manual page (in section 9) for details of this command. However, the default behavior in Windows is to run jobs under a dynamically created dedicated execution account, so just using the default behavior is sufficient to avoid problems with lurker processes.
3. You can tell if the starter is in fact treating the account as a dedicated account, because it will print a line such as the following in its log file:

   ```
   Tracking process family by login "cndrusr1"
   ```

Working Directories for Jobs

Every executing process has a notion of its current working directory. This is the directory that acts as the base for all file system access. There are two current working directories for any Condor job: one where the job is submitted and a second where the job executes. When a user submits a job, the submit-side current working directory is the same as for the user when the *condor_submit* command is issued. The **initialdir** submit command may change this, thereby allowing different jobs to have different working directories. This is useful when submitting large numbers of jobs. This submit-side current working directory remains unchanged for the entire life of a job. The submit-side current working directory is also the working directory of the *condor_shadow* daemon. This is particularly relevant for standard universe jobs, since file system access for the job goes through the *condor_shadow* daemon, and therefore all accesses behave as if they were executing without Condor.

There is also an execute-side current working directory. For standard universe jobs, it is set to the `execute` subdirectory of Condor's home directory. This directory is world-writable, since a Condor job usually runs as user `nobody`. Normally, standard universe jobs would never access this directory, since all I/O system calls are passed back to the *condor_shadow* daemon on the submit machine. In the event, however, that a job crashes and creates a core dump file, the execute-side current working directory needs to be accessible by the job so that it can write the core file. The core file is moved back to the submit machine, and the *condor_shadow* daemon is informed. The *condor_shadow* daemon sends e-mail to the job owner announcing the core file, and provides a pointer to where the core file resides in the submit-side current working directory.

3.6.14 Privilege Separation

Section 3.6.13 discusses why, under most circumstances, it is beneficial to run the Condor daemons as `root`. In situations where multiple users are involved or where Condor is responsible for enforcing a machine owner's policy, running as `root` is the *only* way for Condor to do its job correctly and securely.

Unfortunately, this requirement of running Condor as `root` is at odds with a well-established goal of security-conscious administrators: keeping the amount of software that runs with superuser privileges to a minimum. Condor's nature as a large distributed system that routinely communicates with potentially untrusted components over the network further aggravates this goal.

The privilege separation (PrivSep) effort in Condor aims to minimize the amount of code that needs `root`-level access, while still giving Condor the tools it needs to work properly. Note that PrivSep is currently only available for execute side functionality, and is not implemented on Windows.

In the PrivSep model, all logic in Condor that requires superuser privilege is contained in a small component called the PrivSep Kernel. The Condor daemons execute as an unprivileged account. They explicitly request action from the PrivSep Kernel whenever `root`-level operations are needed.

The PrivSep model then prevents the following attack scenario. In the attack scenario, an attacker has found an exploit in the *condor_startd* that allows for execution of arbitrary code on that daemon's behalf. This gives the attacker `root` access and therefore control over any machine on which the *condor_startd* is running as `root` and the exploit can be exercised. Under the PrivSep model, the *condor_startd* no longer runs as `root`. This prevents the attacker from taking arbitrary action as `root`. Further, limits on requested actions from the PrivSep Kernel contain and restrict the attacker's sphere of influence.

The following section describes the configuration necessary to enable PrivSep for an execute-side Condor installation. After this is a detailed description of the services that the PrivSep Kernel provides to Condor, and how it limits the allowed `root`-level actions.

PrivSep Configuration

The PrivSep Kernel is implemented as two programs: the *condor_root_switchboard* and the *condor_procd*. Both are contained in the `sbin` directory of the Condor distribution. When Condor is running in PrivSep mode, these are to be the only two Condor daemons that run with `root` privilege.

Each of these binaries must be accessible on the file system via a *trusted path*. A trusted path ensures that no user (other than `root`) can alter the binary or path to the binary referred to. To ensure that the paths to these binaries are trusted, use only `root`-owned directories, and set the permissions on these directories to deny write access to all but `root`. The binaries themselves must also be owned by `root` and not writable by any other. The *condor_root_switchboard* program additionally is installed with the setuid bit set. The following command properly sets the permissions on the *condor_root_switchboard* binary:

```
chmod 4755 /opt/condor/release/sbin/condor_root_switchboard
```

The PrivSep Kernel has its own configuration file. This file must be `/etc/condor/privsep_config`. The format of this file is different than a Condor configuration file. It consists of lines with "`key = value`" pairs. Lines with only white space or lines with "#" as the first non-white space character are ignored.

In the PrivSep Kernel configuration file, some configuration settings are interpreted as single values, while others are interpreted as lists. To populate a list with multiple values, use multiple lines with the same key. For example, the following configures the `valid-dirs` setting as a list with two entries:

```
valid-dirs = /opt/condor/execute_1
valid-dirs = /opt/condor/execute_2
```

It is an error to have multiple lines with the same key for a setting that is not interpreted as a list.

Some PrivSep Kernel configuration file settings require a list of UIDs or GIDs, and these allow for a more specialized syntax. User and group IDs can be specified either numerically or textually. Multiple list entries may be given on a single line using the `:` (colon) character as a delimiter. In addition, list entries may specify a range of IDs using a `-` (dash) character to separate the minimum and maximum IDs included. The `*` (asterisk) character on the right-hand side of such a range indicates that the range extends to the maximum possible ID. The following example builds a complex list of IDs:

```
valid-target-uids = nobody : nfsuser1 : nfsuser2
valid-target-uids = condor_run_1 - condor_run_8
valid-target-uids = 800 - *
```

If `condor_run_1` maps to UID 701, and `condor_run_8` maps to UID 708, then this range specifies the 8 UIDs of 701 through 708 (inclusive).

The following settings are required to configure the PrivSep Kernel:

- `valid-caller-uids` and `valid-caller-gids`. These lists specify users and groups that will be allowed to request action from the PrivSep Kernel. The list typically will contain the UID and primary GID that the Condor daemons will run as.
- `valid-target-uids` and `valid-target-gids`. These lists specify the users and groups that Condor will be allowed to act on behalf of. The list will need to include IDs of all users and groups that Condor jobs may use on the given execute machine.
- `valid-dirs`. This list specifies directories that Condor will be allowed to manage for the use of temporary job files. Normally, this will only need to include the value of Condor's `$(EXECUTE)` directory. Any entry in this list must be a trusted path. This means that all components of the path must be directories that are `root`-owned and only writable by `root`. For many sites, this may require a change in ownership and permissions to the `$(LOCAL_DIR)` and `$(EXECUTE)` directories. Note also that the PrivSep Kernel does not have access to Condor's configuration variables, and therefore may not refer to them in this file.
- `procd-executable`. A (trusted) full path to the *condor_procd* executable. Note that the PrivSep Kernel does not have access to Condor's configuration variables, and therefore may not refer to them in this file.

Here is an example of a full `privsep_config` file. This file gives the `condor` account access to the PrivSep Kernel. Condor's use of this execute machine will be restricted to a set of eight dedicated accounts, along with the `users` group. Condor's `$(EXECUTE)` directory and the *condor_procd* executable are also specified, as required.

```
valid-caller-uids = condor
valid-caller-gids = condor
valid-target-uids = condor_run_1 - condor_run_8
valid-target-gids = users : condor_run_1 - condor_run_8
valid-dirs = /opt/condor/local/execute
procd-executable = /opt/condor/release/sbin/condor_procd
```

Once the PrivSep Kernel is properly installed and configured, Condor's configuration must be updated to specify that PrivSep should be used. The Condor configuration variable `PRIVSEP_ENABLED` is a boolean flag serving this purpose. In addition, Condor must be told where the *condor_root_switchboard* binary is located using the `PRIVSEP_SWITCHBOARD` setting. The following example illustrates:

```
PRIVSEP_ENABLED = True
PRIVSEP_SWITCHBOARD = $(SBIN)/condor_root_switchboard
```

Finally, note that while the *condor_procd* is in general an optional component of Condor, it is required when PrivSep is in use. If `PRIVSEP_ENABLED` is `True`, the *condor_procd* will be

used regardless of the USE_PROCD setting. Details on these Condor configuration variables are in section 3.3.28 for PrivSep variables and section 3.3.18 for *condor_procd* variables.

PrivSep Kernel Interface

This section describes the root-enabled operations that the PrivSep Kernel makes available to Condor. The PrivSep Kernel's interface is designed to provide only operations needed by Condor in order to function properly. Each operation is further restricted based on the PrivSep Kernel's configuration settings.

The following list describes each action that can be performed via the PrivSep Kernel, along with the limitations enforced on how it may be used. The terms *valid target users*, *valid target groups*, and *valid directories* refer respectively to the settings for valid-target-uids, valid-target-gids, and valid-dirs from the PrivSep Kernel's configuration.

- *Make a directory as a user.* This operation creates an empty directory, owned by a user. The user must be a valid target user, and the new directory's parent must be a valid directory.
- *Change ownership of a directory tree.* This operation involves recursively changing ownership of all files and subdirectories contained in a given directory. The directory's parent must be a valid directory, and the new owner must either be a valid target user or the user invoking the PrivSep Kernel.
- *Remove a directory tree.* This operation deletes a given directory, including everything contained within. The directory's parent must be a valid directory.
- *Execute a program as a user.* Condor can invoke the PrivSep kernel to execute a program as a valid target user. The user's primary group and any supplemental groups that it is a member of must all be valid target groups. This operation may also include opening files for standard input, output, and error before executing the program.

After launching a program as a valid target user, the PrivSep Kernel allows Condor limited control over its execution. The following operations are supported on a program executed via the PrivSep Kernel:

- *Get resource usage information.* This allows Condor to gather usage statistics such as CPU time and memory image size. This applies to the program's initial process and any of its descendants.
- *Signal the program.* Condor may ask that signals be sent to the program's initial process as a notification mechanism.
- *Suspend and resume the program.* These operations send SIGSTOP or SIGCONT signals to all processes that make up the program.
- *Kill the process and all descendants.* Condor is allowed to terminate the execution of the program or any processes left behind when the program completes.

By sufficiently constraining the valid target accounts and valid directories to which the PrivSep Kernel allows access, the ability of a compromised Condor daemon to do damage can be considerably reduced.

3.6.15 Support for *glexec*

glexec is a tool that provides a sudo-like capability in a grid environment. *glexec* takes an X.509 proxy and a command to run as inputs, and maps the proxy to a local identity (that is, a Unix UID), which it then uses to execute the command. Like the *condor_root_switchboard* command, which provides similar functionality for Condor's PrivSep mode (see section 3.6.14), *glexec* must be installed as a root-owned setuid program. See http://www.nikhef.nl/grid/lcaslcmaps/glexec/ for more information about *glexec*.

Condor can interoperate with *glexec*, using it in a similar way to how the *condor_root_switchboard* is used when running Condor in PrivSep mode. The *condor_starter* uses *glexec* when launching a job, in order to give the job a separate UID from that of the Condor daemons. *glexec* is also used when performing maintenance actions such as cleaning up a job's files and processes, which cannot be done well directly under the Condor daemons' UID due to permissions. A consequence of this type of integration with *glexec* is that the execution of a single Condor job results in several *glexec* invocations, and each must map the proxy to the same UID. It is thus important to ensure that *glexec* is configured to provide this guarantee.

Configuration for *glexec* support is straightforward. The boolean configuration variable `GLEXEC_JOB` must be set `True` on execute machines where *glexec* is to be used. Condor also must be given the full path to the *glexec* binary using the `GLEXEC` configuration variable. Note that Condor must be started as a non-root user when *glexec* is used. This is because when Condor runs as root, it can perform actions as other UIDs arbitrarily, and *glexec*'s services are not needed. Finally, for a job to execute properly in the mode utilizing *glexec*, the job must be submitted with a proxy specified via the **x509userproxy** command in its submit description file, since a proxy is needed as input to *glexec*.

Earlier versions of Condor employed a different form of *glexec* support, where the *condor_starter* daemon ran under the same UID as the job. This feature was enabled using the `GLEXEC_STARTER` configuration variable. This configuration variable is no longer used, and it is replaced by the `GLEXEC_JOB` configuration variable, to enable usage of *glexec*.

3.7 Networking (includes sections on Port Usage, CCB, and GCB)

This section on network communication in Condor discusses which network ports are used, how Condor behaves on machines with multiple network interfaces and IP addresses, and how to facilitate functionality in a pool that spans firewalls and private networks.

The security section of the manual contains some information that is relevant to the discussion of network communication which will not be duplicated here, so please see section 3.6 as well.

Firewalls, private networks, and network address translation (NAT) pose special problems for Condor. There are currently two main mechanisms for dealing with firewalls within Condor:

1. Restrict Condor to use a specific range of port numbers, and allow connections through the firewall that use any port within the range.
2. Use *Condor Connection Brokering* (CCB) or *Generic Connection Brokering* (GCB).

Each method has its own advantages and disadvantages, as described below.

3.7.1 Port Usage in Condor

Default Port Usage

Every Condor daemon listens on a network port for incoming commands. (Using *condor_shared_port*, this port may be shared between multiple daemons.) Most daemons listen on a dynamically assigned port. In order to send a message, Condor daemons and tools locate the correct port to use by querying the *condor_collector*, extracting the port number from the ClassAd. One of the attributes included in every daemon's ClassAd is the full IP address and port number upon which the daemon is listening.

To access the *condor_collector* itself, all Condor daemons and tools must know the port number where the *condor_collector* is listening. The *condor_collector* is the only daemon with a well-known, fixed port. By default, Condor uses port 9618 for the *condor_collector* daemon. However, this port number can be changed (see below).

As an optimization for daemons and tools communicating with another daemon that is running on the same host, each Condor daemon can be configured to write its IP address and port number into a well-known file. The file names are controlled using the `<SUBSYS>_ADDRESS_FILE` configuration variables, as described in section 3.3.5 on page 176.

NOTE: In the 6.6 stable series, and Condor versions earlier than 6.7.5, the *condor_negotiator* also listened on a fixed, well-known port (the default was 9614). However, beginning with version 6.7.5, the *condor_negotiator* behaves like all other Condor daemons, and publishes its own ClassAd to the *condor_collector* which includes the dynamically assigned port the *condor_negotiator* is listening on. All Condor tools and daemons that need to communicate with the *condor_negotiator* will either use the `NEGOTIATOR_ADDRESS_FILE` or will query the *condor_collector* for the *condor_negotiator*'s ClassAd.

Sites that configure any checkpoint servers will introduce other fixed ports into their network. Each *condor_ckpt_server* will listen to 4 fixed ports: 5651, 5652, 5653, and 5654. There is currently no way to configure alternative values for any of these ports.

Using a Non Standard, Fixed Port for the *condor_collector*

By default, Condor uses port 9618 for the *condor_collector* daemon. To use a different port number for this daemon, the configuration variables that tell Condor these communication details are modified. Instead of

```
CONDOR_HOST = machX.cs.wisc.edu
COLLECTOR_HOST = $(CONDOR_HOST)
```

the configuration might be

```
CONDOR_HOST = machX.cs.wisc.edu
COLLECTOR_HOST = $(CONDOR_HOST):9650
```

If a non standard port is defined, the same value of `COLLECTOR_HOST` (including the port) must be used for all machines in the Condor pool. Therefore, this setting should be modified in the global configuration file (`condor_config` file), or the value must be duplicated across all configuration files in the pool if a single configuration file is not being shared.

When querying the *condor_collector* for a remote pool that is running on a non standard port, any Condor tool that accepts the **-pool** argument can optionally be given a port number. For example:

```
% condor_status -pool foo.bar.org:1234
```

Using a Dynamically Assigned Port for the *condor_collector*

On single machine pools, it is permitted to configure the *condor_collector* daemon to use a dynamically assigned port, as given out by the operating system. This prevents port conflicts with other services on the same machine. However, a dynamically assigned port is only to be used on single machine Condor pools, and only if the `COLLECTOR_ADDRESS_FILE` configuration variable has also been defined. This mechanism allows all of the Condor daemons and tools running on the same machine to find the port upon which the *condor_collector* daemon is listening, even when this port is not defined in the configuration file and is not known in advance.

To enable the *condor_collector* daemon to use a dynamically assigned port, the port number is set to 0 in the `COLLECTOR_HOST` variable. The `COLLECTOR_ADDRESS_FILE` configuration variable must also be defined, as it provides a known file where the IP address and port information will be stored. All Condor clients know to look at the information stored in this file. For example:

```
COLLECTOR_HOST = $(CONDOR_HOST):0
COLLECTOR_ADDRESS_FILE = $(LOG)/.collector_address
```

NOTE: Using a port of 0 for the *condor_collector* and specifying a `COLLECTOR_ADDRESS_FILE` only works in Condor version 6.6.8 or later in the 6.6 stable series, and in version 6.7.4 or later in the 6.7 development series. Do not attempt to do this with older versions of Condor.

Configuration definition of COLLECTOR_ADDRESS_FILE is in section 3.3.5 on page 176, and COLLECTOR_HOST is in section 3.3.3 on page 162.

Restricting Port Usage to Operate with Firewalls

If a Condor pool is completely behind a firewall, then no special consideration or port usage is needed. However, if there is a firewall between the machines within a Condor pool, then configuration variables may be set to force the usage of specific ports, and to utilize a specific range of ports.

By default, Condor uses port 9618 for the *condor_collector* daemon, and dynamic (apparently random) ports for everything else. See section 3.7.1, if a dynamically assigned port is desired for the *condor_collector* daemon.

All of the Condor daemons on a machine may be configured to share a single port. See section 3.3.36 for more information.

The configuration variables HIGHPORT and LOWPORT facilitate setting a restricted range of ports that Condor will use. This may be useful when some machines are behind a firewall. The configuration macros HIGHPORT and LOWPORT will restrict dynamic ports to the range specified. The configuration variables are fully defined in section 3.3.6. All of these ports must be greater than 0 and less than 65,536. Note that both HIGHPORT and LOWPORT must be at least 1024 for Condor version 6.6.8. In general, use ports greater than 1024, in order to avoid port conflicts with standard services on the machine. Another reason for using ports greater than 1024 is that daemons and tools are often not run as root, and only root may listen to a port lower than 1024. Also, the range must include enough ports that are not in use, or Condor cannot work.

The range of ports assigned may be restricted based on incoming (listening) and outgoing (connect) ports with the configuration variables IN_HIGHPORT , IN_LOWPORT , OUT_HIGHPORT , and OUT_LOWPORT . See section 3.3.6 for complete definitions of these configuration variables. A range of ports lower than 1024 for daemons running as root is appropriate for incoming ports, but not for outgoing ports. The use of ports below 1024 (versus above 1024) has security implications; therefore, it is inappropriate to assign a range that crosses the 1024 boundary.

NOTE: Setting HIGHPORT and LOWPORT will not automatically force the *condor_collector* to bind to a port within the range. The only way to control what port the *condor_collector* uses is by setting the COLLECTOR_HOST (as described above).

The total number of ports needed depends on the size of the pool, the usage of the machines within the pool (which machines run which daemons), and the number of jobs that may execute at one time. Here we discuss how many ports are used by each participant in the system. This assumes that *condor_shared_port* is not being used. If it *is* being used, then all daemons can share a single incoming port.

The central manager of the pool needs 5 + NEGOTIATOR_SOCKET_CACHE_SIZE ports for daemon communication, where NEGOTIATOR_SOCKET_CACHE_SIZE is specified in the configuration or defaults to the value 16.

Each execute machine (those machines running a *condor_startd* daemon) requires `5 + (5 * number of slots advertised by that machine)` ports. By default, the number of slots advertised will equal the number of physical CPUs in that machine.

Submit machines (those machines running a *condor_schedd* daemon) require `5 + (5 * MAX_JOBS_RUNNING)` ports. The configuration variable `MAX_JOBS_RUNNING` limits (on a per-machine basis, if desired) the maximum number of jobs. Without this configuration macro, the maximum number of jobs that could be simultaneously executing at one time is a function of the number of reachable execute machines.

Also be aware that `HIGHPORT` and `LOWPORT` only impact dynamic port selection used by the Condor system, and they do not impact port selection used by jobs submitted to Condor. Thus, jobs submitted to Condor that may create network connections may not work in a port restricted environment. For this reason, specifying `HIGHPORT` and `LOWPORT` is not going to produce the expected results if a user submits MPI applications to be executed under the parallel universe.

Where desired, a local configuration for machines *not* behind a firewall can override the usage of `HIGHPORT` and `LOWPORT`, such that the ports used for these machines are not restricted. This can be accomplished by adding the following to the local configuration file of those machines *not* behind a firewall:

```
HIGHPORT = UNDEFINED
LOWPORT  = UNDEFINED
```

If the maximum number of ports allocated using `HIGHPORT` and `LOWPORT` is too few, socket binding errors of the form

```
failed to bind any port within <$LOWPORT> - <$HIGHPORT>
```

are likely to appear repeatedly in log files.

Multiple Collectors

This section has not yet been written

Port Conflicts

This section has not yet been written

3.7.2 Reducing Port Usage with the *condor_shared_port* Daemon

The *condor_shared_port* is an optional daemon responsible for creating a TCP listener port shared by all of the Condor daemons for which the configuration variable `USE_SHARED_PORT` is `True`. The *condor_master* will invoke the *condor_shared_port* daemon if it is listed in `DAEMON_LIST`.

The main purpose of the *condor_shared_port* daemon is to reduce the number of ports that must be opened when Condor needs to be accessible through a firewall. This has a greater security benefit than simply reducing the number of open ports. Without the *condor_shared_port* daemon, one can configure Condor to use a range of ports, but since some Condor daemons are created dynamically, this full range of ports will not be in use by Condor at all times. This implies that other non-Condor processes not intended to be exposed to the outside network could unintentionally bind to ports in the range intended for Condor, unless additional steps are taken to control access to those ports. While the *condor_shared_port* daemon is running, it is exclusively bound to its port, which means that other non-Condor processes cannot accidentally bind to that port.

A secondary benefit of the *condor_shared_port* daemon is that it helps address the scalability issues of a submit machine. Without the *condor_shared_port* daemon, approximately 2.1 ephemeral ports per running job are required, and possibly more, depending on the rate of job completion. There are only 64K ports in total, and most standard Unix installations only allocate a subset of these as ephemeral ports. In practice, with long running jobs, and with between 11K and 14K simultaneously running jobs, port exhaustion has been observed in typical Linux installations. After increasing the ephemeral port range as to as many as possible, port exhaustion occurred between 20K and 25K running jobs. Using the *condor_shared_port* daemon, each running job requires fewer, approximately 1.1 ephemeral ports on the submit node, if Condor on the submit node connects directly to Condor on the execute node. If the submit node connects via CCB to the execute node, *no* ports are required per running job; only the one port allocated to the *condor_shared_port* daemon is used.

When CCB is utilized via setting the configuration variable `CCB_ADDRESS` , the *condor_shared_port* daemon registers with the CCB server on behalf of all daemons sharing the port. This means that it is not possible to individually enable or disable CCB connectivity to daemons that are using the shared port; they all effectively share the same setting, and the *condor_shared_port* daemon handles all CCB connection requests on their behalf.

Condor's authentication and authorization steps are unchanged by the use of a shared port. Each Condor daemon continues to operate according to its configured policy. Requests for connections to the shared port are not authenticated or restricted by the *condor_shared_port* daemon. They are simply passed to the requested daemon, which is then responsible for enforcing the security policy.

When the *condor_master* is configured to use the shared port by setting the configuration variable

```
USE_SHARED_PORT = True
```

the *condor_shared_port* daemon is treated specially. A command such as *condor_off*, which shuts down all daemons except for the *condor_master*, will also leave the *condor_shared_port* running. This prevents the *condor_master* from getting into a state where it can no longer receive commands.

The *condor_collector* daemon typically has its own port; it uses 9618 by default. However, it can be configured to use a shared port. Since the address of the *condor_collector* must be set in the configuration file, it is necessary to specify the shared port socket name of the *condor_collector*, so that connections to the shared port that are intended for the *condor_collector* can be forwarded to

it. If the shared port number is 11000, a *condor_collector* address using this shared port could be configured:

```
COLLECTOR_HOST = collector.host.name:11000?sock=collector
```

This configuration assumes that the socket name used by the *condor_collector* is `collector`. The *condor_collector* that runs on `collector.host.name` will automatically choose this socket name if `COLLECTOR_HOST` is configured as in the example above. If multiple *condor_collector* daemons are started on the same machine, the socket name can be explicitly set in the daemon arguments, as in the example:

```
COLLECTOR_ARGS = -sock collector
```

When the *condor_collector* address is a shared port, TCP updates will be automatically used instead of UDP. Under Unix, this means that the *condor_collector* daemon should be configured to have enough file descriptors. See section 3.7.6 for more information on using TCP within Condor.

SOAP commands cannot be sent over a shared port. However, a daemon may be configured to open a fixed, non-shared port, in addition to using a shared port. This is done both by setting `USE_SHARED_PORT = True` and by specifying a fixed port for the daemon using `<SUBSYS>_ARGS = -p <portnum>`.

The TCP connections required to manage standard universe jobs do not make use of shared ports.

3.7.3 Configuring Condor for Machines With Multiple Network Interfaces

Condor can run on machines with multiple network interfaces. Starting with Condor version 6.7.13 (and therefore all Condor 6.8 and more recent versions), new functionality is available that allows even better support for multi-homed machines, using the configuration variable `BIND_ALL_INTERFACES`. A multi-homed machine is one that has more than one NIC (Network Interface Card). Further improvements to this new functionality will remove the need for any special configuration in the common case. For now, care must still be given to machines with multiple NICs, even when using this new configuration variable.

Using BIND_ALL_INTERFACES

Machines can be configured such that whenever Condor daemons or tools call `bind()`, the daemons or tools use all network interfaces on the machine. This means that outbound connections will always use the appropriate network interface to connect to a remote host, instead of being forced to use an interface that might not have a route to the given destination. Furthermore, sockets upon which a daemon listens for incoming connections will be bound to all network interfaces on the machine. This means that so long as remote clients know the right port, they can use any IP address on the machine and still contact a given Condor daemon.

This functionality is on by default. To disable this functionality, the boolean configuration variable BIND_ALL_INTERFACES is defined and set to False:

```
BIND_ALL_INTERFACES = FALSE
```

This functionality has limitations. Here are descriptions of the limitations.

Using all network interfaces does not work with Kerberos. Every Kerberos ticket contains a specific IP address within it. Authentication over a socket (using Kerberos) requires the socket to also specify that same specific IP address. Use of BIND_ALL_INTERFACES causes outbound connections from a multi-homed machine to originate over any of the interfaces. Therefore, the IP address of the outbound connection and the IP address in the Kerberos ticket will not necessarily match, causing the authentication to fail. Sites using Kerberos authentication on multi-homed machines are strongly encouraged not to enable BIND_ALL_INTERFACES, at least until Condor's Kerberos functionality supports using multiple Kerberos tickets together with finding the right one to match the IP address a given socket is bound to.

There is a potential security risk. Consider the following example of a security risk. A multi-homed machine is at a network boundary. One interface is on the public Internet, while the other connects to a private network. Both the multi-homed machine and the private network machines comprise a Condor pool. If the multi-homed machine enables BIND_ALL_INTERFACES, then it is at risk from hackers trying to compromise the security of the pool. Should this multi-homed machine be compromised, the entire pool is vulnerable. Most sites in this situation would run an *sshd* on the multi-homed machine so that remote users who wanted to access the pool could log in securely and use the Condor tools directly. In this case, remote clients do not need to use Condor tools running on machines in the public network to access the Condor daemons on the multi-homed machine. Therefore, there is no reason to have Condor daemons listening on ports on the public Internet, causing a potential security threat.

Up to two IP addresses will be advertised. At present, even though a given Condor daemon will be listening to ports on multiple interfaces, each with their own IP address, there is currently no mechanism for that daemon to advertise all of the possible IP addresses where it can be contacted. Therefore, Condor clients (other Condor daemons or tools) will not necessarily able to locate and communicate with a given daemon running on a multi-homed machine where BIND_ALL_INTERFACES has been enabled.

Currently, Condor daemons can only advertise two IP addresses in the ClassAd they send to their *condor_collector*. One is the public IP address and the other is the private IP address. Condor tools and other daemons that wish to connect to the daemon will use the private IP address if they are configured with the same private network name, and they will use the public IP address otherwise. So, even if the daemon is listening on 3 or more different interfaces, each with a separate IP, the daemon must choose which two IP addresses to advertise so that other daemons and tools can connect to it.

By default, Condor advertises the IP address of the network interface used to contact the *condor_collector* as its public address, since this is the most likely to be accessible to other

processes that query the same *condor_collector*. The NETWORK_INTERFACE configuration variable can be used to specify the public IP address Condor should advertise, and PRIVATE_NETWORK_INTERFACE , along with PRIVATE_NETWORK_NAME can be used to specify the private IP address to advertise.

Sites that make heavy use of private networks and multi-homed machines should consider if using the Condor Connection Broker, CCB, is right for them. More information about CCB and Condor can be found in section 3.7.4 on page 367.

Central Manager with Two or More NICs

Often users of Condor wish to set up compute farms where there is one machine with two network interface cards (one for the public Internet, and one for the private net). It is convenient to set up the head node as a central manager in most cases and so here are the instructions required to do so.

Setting up the central manager on a machine with more than one NIC can be a little confusing because there are a few external variables that could make the process difficult. One of the biggest mistakes in getting this to work is that either one of the separate interfaces is not active, or the host/domain names associated with the interfaces are incorrectly configured.

Given that the interfaces are up and functioning, and they have good host/domain names associated with them here is how to configure Condor:

In this example, farm-server.farm.org maps to the private interface. In the central manager's global (to the cluster) configuration file:

```
CONDOR_HOST = farm-server.farm.org
```

In the central manager's local configuration file:

```
NETWORK_INTERFACE = <IP address of farm-server.farm.org>
NEGOTIATOR = $(SBIN)/condor_negotiator
COLLECTOR = $(SBIN)/condor_collector
DAEMON_LIST = MASTER, COLLECTOR, NEGOTIATOR, SCHEDD, STARTD
```

If the central manager and farm machines are all NT, then only vanilla universe will work now. However, if this is set up for Unix, then at this point, standard universe jobs should be able to function in the pool. But, if UID_DOMAIN is not configured to be homogeneous across the farm machines, the standard universe jobs will run as nobody on the farm machines.

In order to get vanilla jobs and file server load balancing for standard universe jobs working (under Unix), do some more work both in the cluster you have put together and in Condor to make everything work. First, you need a file server (which could also be the central manager) to serve files to all of the farm machines. This could be NFS or AFS, and it does not really matter to Condor. The mount point of the directories you wish your users to use must be the same across all of the

farm machines. Now, configure UID_DOMAIN and FILESYSTEM_DOMAIN to be homogeneous across the farm machines and the central manager. Inform Condor that an NFS or AFS file system exists and that is done in this manner. In the global (to the farm) configuration file:

```
# If you have NFS
USE_NFS = True
# If you have AFS
HAS_AFS = True
USE_AFS = True
# if you want both NFS and AFS, then enable both sets above
```

Now, if the cluster is set up so that it is possible for a machine name to never have a domain name (for example, there is machine name but no fully qualified domain name in /etc/hosts), configure DEFAULT_DOMAIN_NAME to be the domain that is to be added on to the end of the host name.

A Client Machine with Multiple Interfaces

If client machine has two or more NICs, then there might be a specific network interface on which the client machine desires to communicate with the rest of the Condor pool. In this case, the local configuration file for the client should have

```
NETWORK_INTERFACE = <IP address of desired interface>
```

A Checkpoint Server on a Machine with Multiple NICs

If a checkpoint server is on a machine with multiple interfaces, then 2 items must be correct to get things to work:

1. The different interfaces have different host names associated with them.
2. In the global configuration file, set configuration variable CKPT_SERVER_HOST to the host name that corresponds with the IP address desired for the pool. Configuration variable NETWORK_INTERFACE must still be specified in the local configuration file for the checkpoint server.

3.7.4 Condor Connection Brokering (CCB)

Condor Connection Brokering, or CCB, is a way of allowing Condor components to communicate with each other when one side is in a private network or behind a firewall. Specifically, CCB allows communication across a private network boundary in the following scenario: a Condor tool

or daemon (process A) needs to connect to a Condor daemon (process B), but the network does not allow a TCP connection to be created from A to B; it only allows connections from B to A. In this case, B may be configured to register itself with a CCB server that both A and B can connect to. Then when A needs to connect to B, it can send a request to the CCB server, which will instruct B to connect to A so that the two can communicate.

As an example, consider a Condor execute node that is within a private network. This execute node's *condor_startd* is process B. This execute node cannot normally run jobs submitted from a machine that is outside of that private network, because bi-directional connectivity between the submit node and the execute node is normally required. However, if both execute and submit machine can connect to the CCB server, if both are authorized by the CCB server, and if it is possible for the execute node within the private network to connect to the submit node, then it is possible for the submit node to run jobs on the execute node.

To effect this CCB solution, the execute node's *condor_startd* within the private network registers itself with the CCB server by setting the configuration variable `CCB_ADDRESS` . The submit node's *condor_schedd* communicates with the CCB server, requesting that the execute node's *condor_startd* open the TCP connection. The CCB server forwards this request to the execute node's *condor_startd*, which opens the TCP connection. Once the connection is open, bi-directional communication is enabled.

If the location of the execute and submit nodes is reversed with respect to the private network, the same idea applies: the submit node within the private network registers itself with a CCB server, such that when a job is running and the execute node needs to connect back to the submit node (for example, to transfer output files), the execute node can connect by going through CCB to request a connection.

If both A and B are in separate private networks, then CCB alone cannot provide connectivity. However, if an incoming port or port range can be opened in one of the private networks, then the situation becomes equivalent to one of the scenarios described above and CCB can provide bi-directional communication given only one-directional connectivity. See section for information on opening port ranges. Also note that CCB works nicely with *condor_shared_port*.

Unfortunately at this time, CCB does not support standard universe jobs.

Any *condor_collector* may be used as a CCB server. There is no requirement that the *condor_collector* acting as the CCB server be the same *condor_collector* that a daemon advertises itself to (as with `COLLECTOR_HOST`). However, this is often a convenient choice.

Example Configuration

This example assumes that there is a pool of machines in a private network that need to be made accessible from the outside, and that the *condor_collector* (and therefore CCB server) used by these machines is accessible from the outside. Accessibility might be achieved by a special firewall rule for the *condor_collector* port, or by being on a dual-homed machine in both networks.

The configuration of variable `CCB_ADDRESS` on machines in the private network causes regis-

tration with the CCB server as in the example:

```
CCB_ADDRESS = $(COLLECTOR_HOST)
PRIVATE_NETWORK_NAME = cs.wisc.edu
```

The definition of PRIVATE_NETWORK_NAME ensures that all communication between nodes within the private network continues to happen as normal, and without going through the CCB server. The name chosen for PRIVATE_NETWORK_NAME should be different from the private network name chosen for any Condor installations that will be communicating with this pool.

Under Unix, and with large Condor pools, it is also necessary to give the *condor_collector* acting as the CCB server a large enough limit of file descriptors. This may be accomplished with the configuration variable MAX_FILE_DESCRIPTORS or an equivalent. Each Condor process configured to use CCB with CCB_ADDRESS requires one persistent TCP connection to the CCB server. A typical execute node requires one connection for the *condor_master*, one for the *condor_startd*, and one for each running job, as represented by a *condor_starter*. A typical submit machine requires one connection for the *condor_master*, one for the *condor_schedd*, and one for each running job, as represented by a *condor_shadow*. If there will be no administrative commands required to be sent to the *condor_master* from outside of the private network, then CCB may be disabled in the *condor_master* by assigning MASTER.CCB_ADDRESS to nothing:

```
MASTER.CCB_ADDRESS =
```

Completing the count of TCP connections in this example: suppose the pool consists of 500 8-slot execute nodes and CCB is not disabled in the configuration of the *condor_master* processes. In this case, the count of needed file descriptors plus some extra for other transient connections to the collector is 500*(1+1+8)=5000. Be generous, and give it twice as many descriptors as needed by CCB alone:

```
COLLECTOR.MAX_FILE_DESCRIPTORS = 10000
```

Security and CCB

The CCB server authorizes all daemons that register themselves with it (using CCB_ADDRESS) at the DAEMON authorization level (these are playing the role of process A in the above description). It authorizes all connection requests (from process B) at the READ authorization level. As usual, whether process B authorizes process A to do whatever it is trying to do is up to the security policy for process B; from the Condor security model's point of view, it is as if process A connected to process B, even though at the network layer, the reverse is true.

Troubleshooting CCB

Errors registering with CCB or requesting connections via CCB are logged at level D_ALWAYS in the debugging log. These errors may be identified by searching for "CCB" in the log message.

Command-line tools require the argument **-debug** for this information to be visible. To see details of the CCB protocol add `D_FULLDEBUG` to the debugging options for the particular Condor subsystem of interest. Or, add `D_FULLDEBUG` to `ALL_DEBUG` to get extra debugging from all Condor components.

A daemon that has successfully registered itself with CCB will advertise this fact in its address in its ClassAd. The ClassAd attribute `MyAddress` will contain information about its `"CCBID"`.

Scalability and CCB

Any number of CCB servers may be used to serve a pool of Condor daemons. For example, half of the pool could use one CCB server and half could use another. Or for redundancy, all daemons could use both CCB servers and then CCB connection requests will load-balance across them. Typically, the limit of how many daemons may be registered with a single CCB server depends on the authentication method used by the *condor_collector* for DAEMON-level and READ-level access, and on the amount of memory available to the CCB server. We are not able to provide specific recommendations at this time, but to give a very rough idea, a server class machine should be able to handle CCB service plus normal *condor_collector* service for a pool containing a few thousand slots without much trouble.

3.7.5 Generic Connection Brokering (GCB)

At this time, the functionality of GCB is being replaced by CCB. Therefore, consider using CCB instead if it provides the needed services. The functionality that GCB provides (over CCB) is communication between two different private networks. CCB only supports communication between nodes with one-directional connectivity. The main reasons why CCB is preferable are: support for all platforms (including Windows), easier configuration and troubleshooting, and ability to restart and reconfigure on the fly.

Generic Connection Brokering, or GCB, is a system for managing network connections across private network and firewall boundaries. Condor's Linux releases are linked with GCB, and can use GCB functionality to run jobs (either directly or via flocking) on pools that span public and private networks.

While GCB provides numerous advantages over restricting Condor to use a range of ports which are then opened on the firewall (see section 3.7.1 on page 361), GCB is also a very complicated system, with major implications for Condor's networking and security functionality. Therefore, sites must carefully weigh the advantages and disadvantages of attempting to configure and use GCB before making a decision.

Advantages:

- Better connectivity. GCB works with pools that have multiple private networks (even multiple private networks that use the same IP addresses (for example, 192.168.2.*). GCB also works with sites that use network address translation (NAT).

- More secure. Administrators never need to allow inbound connections through the firewall. With GCB, only outbound connections from behind the firewall must be allowed (which is a standard firewall configuration). It is possible to trade decreased performance for better security, and configure the firewall to only allow outbound connections to a single public IP address.
- Does not require `root` access to any machines. All parts of a GCB system can be run as an unprivileged user, and in the common case, no changes to the firewall configuration are required.

Disadvantages:

- The GCB broker (section 3.7.5 describes the broker) node(s) is a potential failure point to the pool. Any private nodes that want to communicate outside their own network must be represented by a GCB broker. This machine must be highly reliable, since if the broker is ever down, all inbound communication with the private nodes is impossible. Furthermore, no other Condor services should be run on a GCB broker (for example, the Condor pool's central manager). While it is possible to do so, it is not recommended. In general, no other services should be run on the machine at all, and the host should be dedicated to the task of serving as a GCB broker.
- All Condor nodes behind a given firewall share a single IP address (the public IP address of their GCB broker). All Condor daemons using a GCB broker will advertise themselves with this single IP address, and in some cases, connections to/from those daemons will actually originate at the broker. This has implications for Condor's host/IP based security, and the general level of confusion for users and administrators of the pool. Debugging problems will be more difficult, as any log messages which only print the IP address (not the name and/or port) will become ambiguous. Even log or error messages that include the port will not necessarily be helpful, as it is difficult to correlate ports on the broker with the corresponding private nodes.
- Can not function with Kerberos authentication. Kerberos tickets include the IP address of the machine where they were created. However, when Condor daemons are using GCB, they use a different IP address, and therefore, any attempt to authenticate using Kerberos will fail, as Kerberos will consider this a (poor) attempt to fool it into using an invalid host principle.
- Scalability and performance degradation:
  - Connections are more expensive to establish.
  - In some cases, connections must be forwarded through a proxy server on the GCB broker.
  - Each network port on each private node must correspond to a unique port on the broker host, so there is a fixed limit to how many private nodes a given broker can service (which is a function of the number of ports each private node requires and the total number of available ports on the broker).

  - Each private node must maintain an open TCP connection to its GCB broker. GCB will attempt to recover in the case of the socket being closed, but this means the broker must have at least as many sockets open as there are private nodes.

- It is more complex to configure and debug.

Given the increased complexity, use of GCB requires a careful read of this entire manual section, followed by a thorough installation.

Details of GCB and how it works can be found at the GCB homepage:

http://www.cs.wisc.edu/condor/gcb

This information is useful for understanding the technical details of how GCB works, and the various parts of the system. While some of the information is partly out of date (especially the discussion of how to configure GCB) most of the sections are perfectly accurate and worth reading. Ignore the section on "GCBnize", which describes how to get a given application to use GCB, as the Linux port of all Condor daemons and tools have already been converted to use GCB.

The rest of this section gives the details for configuring a Condor pool to use GCB. It is divided into the following topics:

- Introduction to the GCB broker
- Configuring the GCB broker
- Spawning a GCB broker (with a *condor_master* or using *initd*)
- How to configure Condor machines to use GCB
- Configuring the GCB routing table
- Implications for Condor's host/IP security settings
- Implications for other Condor configuration settings

Introduction to the GCB Broker

At the heart of GCB is a logical entity known as a *broker* or *inagent*. In reality, the entity is made up of daemon processes running on the same machine comprised of the *gcb_broker* and a set of *gcb_relay_server* processes, each one spawned by the *gcb_broker*.

Every private network using GCB must have at least one broker to arrange connections. The broker must be installed on a machine that nodes in both the public and the private (firewalled) network can directly talk to. The broker need not be able to initiate connections to the private nodes. It can take advantage of the case where it can initiate connections to the private nodes, and that will improve performance. The broker is generally installed on a machine with multiple network interfaces (on the network boundary) or just outside of a network that allows outbound connections.

If the private network contains many hosts, sites can configure multiple GCB brokers, and partition the private nodes so that different subsets of the nodes use different brokers.

For a more thorough explanation of what a GCB broker is, check out: http://www.cs.wisc.edu/~sschang/firewall/gcb/mechanism.htm

A GCB broker should generally be installed on a dedicated machine. These are machines that are not running other Condor daemons or services. If running any other Condor service (for example, the central manager of the pool) on the same machine as the GCB broker, all other machines attempting to use this Condor service (for example, to connect to the *condor_collector* or *condor_negotiator*) will incur additional connection costs and latency. It is possible that future versions of GCB and Condor will be able to overcome these limitations, but for now, we recommend that a broker is run on a dedicated machine with no other Condor daemons (except perhaps a single *condor_master* used to spawn the *gcb_broker* daemon, as described below).

In principle, a GCB broker is a network element that functions almost like a router. It allows certain connections through the firewall by redirecting connections or forwarding connections. In general, it is not a good idea to run a lot of other services on the network elements, especially not services like Condor which can spawn arbitrary jobs. Furthermore, the GCB broker relies on listening to many network ports. If other applications are running on the same host as the broker, problems exist where the broker does not have enough network ports available to forward all the connections that might be required of it. Also, all nodes inside a private network rely on the GCB broker for all incoming communication. For performance reasons, avoid forcing the GCB broker to contend with other processes for system resources, such that it is always available to handle communication requests. There is nothing in GCB or Condor requiring the broker to run on a separate machine, but it is the recommended configuration.

The *gcb_broker* daemon listens on two hard-coded, fixed ports (65432 and 65430). A future version of Condor and GCB will remove this limitation. However, for now, to run a *gcb_broker* on a given host, ensure that ports 65432 and 65430 are not already in use.

If `root` access on a machine where a GCB broker is planned, one good option is to have *initd* configured to spawn (and re-spawn) the *gcb_broker* binary (which is located in the <`release_dir`>`/libexec` directory). This way, the *gcb_broker* will be automatically restarted on reboots, or in the event that the broker itself crashes or is killed. Without `root` access, use a *condor_master* to manage the *gcb_broker* binary.

Configuring the GCB broker

Since the *gcb_broker* and *gcb_relay_server* are not Condor daemons, they do not read the Condor configuration files. Therefore, they must be configured by other means, namely the environment and through the use of command-line arguments.

There is one required command-line argument for the *gcb_broker*. This argument defines the public IP address this broker will use to represent itself and any private network nodes that are configured to use this broker. This information is defined with **-i xxx.xxx.xxx.xxx** on the command-line when the *gcb_broker* is executed. If the broker is being setup outside the private network, it is

likely that the machine will only have one IP address, which is clearly the one to use. However, if the broker is being run on a machine on the network boundary (a multi-homed machine with interfaces into both the private and public networks), be sure to use the IP address of the interface on the public network.

Additionally, specify environment variables to control how the *gcb_broker* (and the *gcb_relay_server* processes it spawns) will behave. Some of these settings can also be specified as command-line arguments to the *gcb_broker*. All of them have reasonable defaults if not defined.

- General daemon behavior

  The environment variable `GCB_RELAY_SERVER` defines the full path to the `gcb_relay_server` binary the broker should use. The command-line override for this is **-r /full/path/to/relayserver**. If not set either on the command-line or in the environment, the *gcb_broker* process will search for a program named `gcb_relay_server` in the same directory where the `gcb_broker` binary is located, and attempt to use that one.

  The environment variable `GCB_ACTIVE_TO_CLIENT` is a boolean that defines whether the GCB broker can directly talk to servers running inside the network that it manages The value must be `yes` or `no`, case sensitive. `GCB_ACTIVE_TO_CLIENT` should be set to `yes` only if this GCB broker is running on a network boundary and can connect to both the private and public nodes. If the broker is running in the public network, it should be left undefined or set to `no`.

- Log file locations

  The environment variable `GCB_LOG_DIR` defines a directory to use for all GCB-related log files. If defined, and the per-daemon log file settings (described below) are not defined, the broker will write to `$GCB_LOG_DIR/BrokerLog` and the relay server will write to `$GCB_LOG_DIR/RelayServerLog.<pid>`

  The environment variable `GCB_BROKER_LOG` defines the full path for the GCB broker's log file. The command-line override is **-l /full/path/to/log/file**. This definition overrides `GCB_LOG_DIR`.

  The environment variable `GCB_RELAY_SERVER_LOG` defines the full path to the GCB relay server's log file. Each relay server writes its own log file, so the actual filename will be: `$GCB_RELAY_SERVER_LOG.<pid>` where `<pid>` is replaced with the process id of the corresponding *gcb_relay_server*. When defined, this setting overrides `GCB_LOG_DIR`.

- Verbose logging

  The environment variable `GCB_DEBUG_LEVEL` controls how verbose all the GCB daemon's log files should be. Can be either `fulldebug` (more verbose) or `basic`. This defines logging behavior for all GCB daemons, unless the following daemon-specific settings are defined.

The environment variable GCB_BROKER_DEBUG controls verbose logging specifically for the GCB broker. The command-line override for this is **-d level**. Overrides GCB_DEBUG_LEVEL.

The environment variable GCB_RELAY_SERVER_DEBUG controls verbose logging specifically for the GCB relay server. Overrides GCB_DEBUG_LEVEL.

- Maximum log file size

The environment variable GCB_MAX_LOG defines the maximum size in bytes of all GCB log files. When the log file reaches this size, the content of the file will be moved to filename.old, and a new log is started. This defines logging behavior for all GCB daemons, unless the following daemon-specific settings are used.

The environment variable GCB_BROKER_MAX_LOG defines the maximum size in bytes of the GCB broker log file.

The environment variable GCB_RELAY_SERVER_MAX_LOG defines the maximum size in bytes of the GCB relay server log file.

Spawning the GCB Broker

There are two ways to spawn the GCB broker:

- Use a *condor_master*.

To spawn the GCB broker with a *condor_master*, here are the recommended condor_config settings that will work:

```
# Specify that you only want the master and the broker running
DAEMON_LIST = MASTER, GCB_BROKER

# Define the path to the broker binary for the master to spawn
GCB_BROKER = $(RELEASE_DIR)/libexec/gcb_broker

# Define the path to the release_server binary for the broker to use
GCB_RELAY = $(RELEASE_DIR)/libexec/gcb_relay_server

# Setup the gcb_broker's environment.  We use a macro to build up the
# environment we want in pieces, and then finally define
# GCB_BROKER_ENVIRONMENT, the setting that condor_master uses.

# Initialize an empty macro
GCB_BROKER_ENV =

# (recommended) Provide the full path to the gcb_relay_server
GCB_BROKER_ENV = $(GCB_BROKER_ENV);GCB_RELAY_SERVER=$(GCB_RELAY)

# (recommended) Tell GCB to write all log files into the Condor log
# directory (the directory used by the condor_master itself)
GCB_BROKER_ENV = $(GCB_BROKER_ENV);GCB_LOG_DIR=$(LOG)
# Or, you can specify a log file separately for each GCB daemon:
```

```
#GCB_BROKER_ENV = $(GCB_BROKER_ENV);GCB_BROKER_LOG=$(LOG)/GCB_Broker_Log
#GCB_BROKER_ENV = $(GCB_BROKER_ENV);GCB_RELAY_SERVER_LOG=$(LOG)/GCB_RS_Log

# (optional -- only set if true) Tell the GCB broker that it can
# directly connect to machines in the private network which it is
# handling communication for.  This should only be enabled if the GCB
# broker is running directly on a network boundary and can open direct
# connections to the private nodes.
#GCB_BROKER_ENV = $(GCB_BROKER_ENV);GCB_ACTIVE_TO_CLIENT=yes

# (optional) turn on verbose logging for all of GCB
#GCB_BROKER_ENV = $(GCB_BROKER_ENV);GCB_DEBUG_LEVEL=fulldebug
# Or, you can turn this on separately for each GCB daemon:
#GCB_BROKER_ENV = $(GCB_BROKER_ENV);GCB_BROKER_DEBUG=fulldebug
#GCB_BROKER_ENV = $(GCB_BROKER_ENV);GCB_RELAY_SERVER_DEBUG=fulldebug

# (optional) specify the maximum log file size (in bytes)
#GCB_BROKER_ENV = $(GCB_BROKER_ENV);GCB_MAX_LOG=640000
# Or, you can define this separately for each GCB daemon:
#GCB_BROKER_ENV = $(GCB_BROKER_ENV);GCB_BROKER_MAX_LOG=640000
#GCB_BROKER_ENV = $(GCB_BROKER_ENV);GCB_RELAY_SERVER_MAX_LOG=640000

# Finally, set the value the condor_master really uses
GCB_BROKER_ENVIRONMENT = $(GCB_BROKER_ENV)

# If your Condor installation on this host already has a public
# interface as the default (either because it is the first interface
# listed in this machine's host entry, or because you've already
# defined NETWORK_INTERFACE), you can just use Condor's special macro
# that holds the IP address for this.
GCB_BROKER_IP = $(ip_address)
# Otherwise, you could define it yourself with your real public IP:
# GCB_BROKER_IP = 123.123.123.123

# (required) define the command-line arguments for the broker
GCB_BROKER_ARGS = -i $(GCB_BROKER_IP)
```

Once those settings are in place, either spawn or restart the *condor_master* and the *gcb_broker* should be started. Ensure the broker is running by reading the log file specified with `GCB_BROKER_LOG`, or in `$(LOG)/BrokerLog` if using the default.

- Use *initd*.

  The system's *initd* may be used to manage the *gcb_broker* without running the *condor_master* on the broker node, but this requires `root` access. Generally, this involves adding a line to the `/etc/inittab` file. Some sites use other means to manage and generate the `/etc/inittab`, such as *cfengine* or other system configuration management tools, so check with the local system administrator to be sure. An example line might be something like:

  ```
  GB:23:respawn:/path/to/gcb_broker -i 123.123.123.123 -r /path/to/relay_server
  ```

  It may be easier to wrap the *gcb_broker* binary in a shell script, in order to change the command-line arguments (and set environment variables) without having to edit `/etc/inittab` all the time. This will be similar to:

```
GB:23:respawn:/opt/condor-6.7.13/libexec/gcb_broker.sh
```

Then, create the wrapper, as similar to:

```
#!/bin/sh

libexec=/opt/condor-6.7.13/libexec
ip=123.123.123.123
relay=$libexec/gcb_relay_server

exec $libexec/gcb_broker -i $ip -r $relay
```

You will probably also want to set some environment variables to tell the GCB daemons where to write their log files (GCB_LOG_DIR), and possibly some of the other variables described above.

Either way, after updating the /etc/inittab, send the *initd* process (always PID 1) a SIGHUP signal, and it will re-read the inittab and spawn the *gcb_broker*.

Configuring Condor nodes to be GCB clients

In general, before configuring a node in a Condor pool to use GCB, the GCB broker node(s) for the pool must be set up and running. Set up, configure, and spawn the broker first.

To enable the use of GCB on a given Condor host, set the following Condor configuration variables:

```
# Tell Condor to use a network remapping service (currently only GCB
# is supported, but in the future, there might be other options)
NET_REMAP_ENABLE = true
NET_REMAP_SERVICE = GCB
```

Only GCB clients within a private network need to define the following variable, which specifies the IP addresses of the brokers serving this network. Note that these IP addresses must match the IP address that was specified on each broker's command-line with the **-i** option.

```
# Public IP address (in standard dot notation) of the GCB broker(s)
# serving this private node.
NET_REMAP_INAGENT = xxx.xxx.xxx.xxx, yyy.yyy.yyy.yyy
```

When more than one IP address is given, the *condor_master* picks one at random for it and all of its descendants to use. Because the NET_REMAP_INAGENT setting is only valid on private nodes, it should not be defined in a global Condor configuration file (condor_config) if the pool also contains nodes on a public network.

Finally, if setting up the recommended (but optional) GCB routing table, tell Condor daemons where to find their table. Define the following variable:

```
# The full path to the routing table used by GCB
NET_REMAP_ROUTE = /full/path/to/GCB-routing-table
```

Setting NET_REMAP_ENABLE causes the BIND_ALL_INTERFACES variable to be automatically set. More information about this setting can be found in section 3.7.3 on page 364. It would not hurt to place the following in the configuration file near the other GCB-related settings, just to remember it:

```
# Tell Condor to bind to all network interfaces, instead of a single
# interface.
BIND_ALL_INTERFACES = true
```

Once a GCB broker is set up and running to manage connections for each private network, and the Condor installation for all the nodes in either private and public networks are configured to enable GCB, restart the Condor daemons, and all of the different machines should be able to communicate with each other.

Configuring the GCB routing table

By default, a GCB-enabled application will always attempt to directly connect to a given IP/port pair. In the case of a private nodes being represented by a GCB broker, the IP/port will be a proxy socket on the broker node, not the real address at each private node. When the GCB broker receives a direct connection to one of its proxy sockets, it notifies the corresponding private node, which establishes a new connection to the broker. The broker then forwards packets between these two sockets, establishing a communication pathway into the private node. This allows clients which are not linked with the GCB libraries to communicate with private nodes using a GCB broker.

This mechanism is expensive in terms of latency (time between messages) and total bandwidth (how much data can be moved in a given time period), as well as expensive in terms of the broker's system resources such as network I/O, processor time, and memory. This expensive mechanism is unnecessary in the case of GCB-aware clients trying to connect to private nodes that can directly communicate with the public host. The alternative is to contact the GCB broker's command interface (the fixed port where the broker is listening for GCB management commands), and use a GCB-specific protocol to request a connection to the given IP/port. In this case, the GCB broker will notify the private node to directly connect to the public client (technically, to a new socket created by the GCB client library linked in with the client's application), and a direct socket between the two is established, removing the need for packet forwarding between the proxy sockets at the GCB broker.

On the other hand, in cases where a direct connection from the client to a given server is possible (for example, two GCB-aware clients in the same public network attempting to communicate with each other), it is expensive and unnecessary to attempt to contact a GCB broker, and the client should connect directly.

To allow a GCB-enabled client to know if it should make a direct connection (which might involve packet forwarding through proxy sockets), or if it should use the GCB protocol to commu-

nicate with the broker's command port and arrange a direct socket, GCB provides a *routing table*. Using this table, an administrator can define what IP addresses should be considered private nodes where the GCB connection protocol will be used, and what nodes are public, where a direct connection (without incurring the latency of contacting the GCB broker, only to find out there is no information about the given IP/port) should be made immediately.

If the attempt to contact the GCB broker for a given IP/port fails, or if the desired port is not being managed by the broker, the GCB client library making the connection will fall back and attempt a direct connection. Therefore, configuring a GCB routing table is not required for communication to work within a GCB-enabled environment. However, the GCB routing table can significantly improve performance for communication with private nodes being represented by a GCB broker.

One confusing aspect of GCB is that all of the nodes on a private network believe that their own IP address is the address of their GCB broker. Due to this, all the Condor daemons on a private network advertise themselves with the same IP address (though the broker will map the different ports to different nodes within the private network). Therefore, a given node in the public network needs to be told that if it is contacting this IP address, it should know that the IP address is really a GCB broker representing a node in the private network, so that the public network node can contact the broker to arrange a single socket from the private node to the public one, instead of relying on forwarding packets between proxy sockets at the broker. Any other addresses, such as other public IP addresses, can be contacted directly, without going through a GCB broker. Similarly, other nodes within the same private network will still be advertising their address with their GCB broker's public IP address. So, nodes within the same private network also have to know that the public IP address of the broker is really a GCB broker, yet all other public IP addresses are valid for direct communication.

In general, all connections can be made directly, except to a host represented by a GCB broker. Furthermore, the default behavior of the GCB client library is to make a direct connection. The routing table is a (somewhat complicated) way to tell a given GCB installation what GCB brokers it might have to communicate with, and that it should directly communicate with anything else. In practice, the routing table should have a single entry for each GCB broker in the system. Future versions of GCB will be able to make use of more complicated routing behavior, which is why the full routing table infrastructure described below is implemented, even if the current version of GCB is not taking advantage of all of it.

Format of the GCB routing table

The routing table is a plain ASCII text file. Each line of the file contains one rule. Each rule consists of a *target* and a *method*. The target specifies destination IP address(es) to match, and the method defines what mechanism must be used to connect to the given target. The target must be a valid IP address string in the standard dotted notation, followed by a slash character (`/`), as well as an integer *mask*. The mask specifies how many bits of the destination IP address and target IP address must match. The method must be one of the strings

```
GCB
direct
```

GCB stops searching the table as soon as it finds a matching rule, therefore place more specific rules (rules with a larger value for the mask and without wild cards) before generic rules (rules with wild cards or smaller mask values). The default when no rule is matched is to use direct communication. Some examples and the corresponding routing tables may help clarify this syntax.

Simple GCB routing table example (1 private, 1 public)

Consider an example with a private network that has a set of nodes whose IP addresses are `192.168.2.*`. Other nodes are in a public network whose IP addresses are `123.123.123.*`. A GCB broker for the 192 network is running on IP address `123.123.123.123`. In this case, the routing table for both the public and private nodes should be:

```
123.123.123.123/32 GCB
```

This rule states that for IP addresses where all 32 bits exactly match the address `123.123.123.123`, first communicate with the GCB broker.

Since the default is to directly connect when no rule in the routing table matches a given target IP, this single rule is all that is required. However, to illustrate how the routing table syntax works, the following routing table is equivalent:

```
123.123.123.123/32 GCB
*/0 direct
```

Any attempt to connect to `123.123.123.123` uses GCB, as it is the first rule in the file. All other IP addresses will connect directly. This table explicitly defines GCB's default behavior.

More complex GCB routing table example (2 private, 1 public)

As a more complicated case, consider a single Condor pool that spans one public network and two private networks. The two separate private networks each have machines with private addresses like `192.168.2.*`. Identify one of these private networks as `A`, and the other one as `B`. The public network has nodes with IP addresses like `123.123.123.*`. Assume that the GCB broker for nodes in the `A` network has IP address `123.123.123.65`, and the GCB broker for the nodes in the `B` network has IP address `123.123.123.66`. All of the nodes need to be able to talk to each other. In this case, nodes in private network `A` advertise themselves as `123.123.123.65`, so any node, regardless of being in A, B, or the public network, must treat that IP address as a GCB broker. Similarly, nodes in private network `B` advertise themselves as `123.123.123.66`, so any node, regardless of being in A, B, or the public network, must treat that IP address as a GCB broker. All other connections from any node can be made directly. Therefore, here is the appropriate routing table for all nodes:

```
123.123.123.65/32 GCB
123.123.123.66/32 GCB
```

Implications of GCB on Condor's Host/IP-based Security Configuration

When a message is received at a Condor daemon's command socket, Condor authenticates based on the IP address of the incoming socket. For more information about this host-based security in Condor, see section 3.6.9 on page 342. Because of the way GCB changes the IP addresses that are used and advertised by GCB-enabled clients, and since all nodes being represented by a GCB broker are represented by different ports on the broker node (a process known as *address leasing*), using GCB has implications for this process.

Depending on the communication pathway used by a GCB-enabled Condor client (either a tool or another Condor daemon) to connect to a given Condor server daemon, and where in the network each side of the connection resides, the IP address of the resulting socket actually used will be very different. In the case of a private client (that is, a client behind a firewall, which may or may not be using NAT and a fully private, non-routable IP address) attempting to connect to a server, there are three possibilities:

- For a direct connection to another node within the private network, the server will see the private IP address of the client.
- For a direct outbound connection to a public node: if NAT is being used, the server will see the IP address of the NAT server for the private network. If there is no NAT, and the firewall is blocking connections in only one direction, but not re-writing IP addresses, the server will see the client's real IP address.
- For a connection to a host in a different private network that must be relayed through the GCB broker, the server will see the IP address of the GCB broker representing the server. This is an instance of the private server case, as described below.

Therefore, any public server that wants to allow a command from a specific client must have any or all of the various IP addresses mentioned above within the appropriate `HOSTALLOW` settings. In practice, that means opening up the `HOSTALLOW` settings to include not just the actual IP addresses of each node, but also the IP address of the various GCB brokers in use, and potentially, the public IP address of the NAT host for each private network.

However, given that all private nodes which are represented by a given GCB broker could potentially make connections to any other host using the GCB broker's IP address (whenever proxy socket forwarding is being used), if a single private node is being granted a certain level of permission within the Condor pool, all of the private nodes using the same GCB broker will have the same level of permission. This is particularly important in the consideration of granting `HOSTALLOW_ADMINISTRATOR` or `HOSTALLOW_CONFIG` privileges to a private node represented by a GCB broker.

In the case of a public client attempting to connect to a private server, there are only two possible cases:

- the GCB broker can arrange a direct socket from the private server. The private server will see the real public IP address of the client.

- the GCB broker must forward packets from a proxy socket. This may happen because of a non-GCB aware public client, a misconfigured or missing GCB routing table, or a client in a different private network. The private server will see the IP address of its own GCB broker. In the case where the GCB broker runs on a node on the network boundary, the private server will see the GCB broker's private IP address (even if the GCB broker is also listening on the public interface and the leased addresses it provides use the public IP addresses). If the GCB broker is running entirely in the public network and cannot directly connect to the private nodes, the private server will see the remote connection as coming from the broker's public IP address.

This second case is particularly troubling. Since there are legitimate circumstances where a private server would need to use a forwarded proxy socket from its GCB broker, in general, the server should allow requests originating from its GCB broker. But, precisely because of the proxy forwarding, that implies that *any* client that can connect to the GCB broker would be allowed into the private server (if IP-based authorization was the only defense).

The final host-based security setting that requires special mention is `HOSTALLOW_NEGOTIATOR` . If the *condor_negotiator* for the pool is running on a private node being represented by a GCB broker, there must be modifications to the default value. For the purposes of Condor's host-based security, the *condor_negotiator* acts as a client when communicating with each *condor_schedd* in the pool which has idle jobs that need to be matched with available resources. Therefore, all the possible cases of a private client attempting to connect to a given server apply to a private *condor_negotiator*. In practice, that means adding the public IP address of the broker, the real private IP address of the negotiator host, and possibly the public IP address of the NAT host for this private network to the `HOSTALLOW_NEGOTIATOR` setting. Unfortunately, this implies that *any* host behind the same NAT host or using the same GCB broker will be authorized as if it was the *condor_negotiator*.

Future versions of GCB and Condor will hopefully add some form of authentication and authorization to the GCB broker itself, to help alleviate these problems. Until then, sites using GCB are encouraged to use GSI strong authentication (since Kerberos also depends on IP addresses and is therefore incompatible with GCB) to rely on an authorization system that is not affected by address leasing. This is especially true for sites that (foolishly) choose to run their central manager on a private node.

Implications of GCB for Other Condor Configuration

Using GCB and address leasing has implications for Condor configuration settings outside of the Host/IP-based security settings. Each is described.

`COLLECTOR_HOST` If the *condor_collector* for the pool is running on a private node being represented by a GCB broker, `COLLECTOR_HOST` must be set to the host name or IP address of the GCB broker machine, *not* the real host name/IP address of the private node where the daemons are actually running. When the *condor_collector* on the private node attempts to `bind()` to its command port (9618 by default), it will request port 9618 on the GCB broker

node, instead. The port is not a worry, but the host name or IP address is a worry. When public nodes want to communicate with the *condor_collector*, they must go through the GCB broker. In theory, other nodes inside the same private network could be told to directly use the private IP address of the *condor_collector* host, but that is unnecessary, and would probably lead to other confusion and configuration problems.

However, because the *condor_collector* is listening on a fixed port, and that single port is reserved on the GCB broker node, no two private nodes using the same broker can attempt to use the same port for their *condor_collector*. Therefore, any site that is attempting to set up multiple pools within the same private network is strongly encouraged to set up separate GCB brokers for each pool. Otherwise, one or both of the pools must use a non-standard port for the *condor_collector*, which adds yet more complication to an already complicated situation.

CKPT_SERVER_HOST Much like the case for COLLECTOR_HOST described above, a checkpoint server on a private node will have to lease a port on the GCB broker node. However, the checkpoint server also uses a fixed port, and unlike the *condor_collector*, there is no way to configure an alternate value. Therefore, only a single checkpoint server can be run behind a given GCB broker. The same solution works: if multiple checkpoint servers are required, multiple GCB brokers are deployed and configured. Furthermore, the host name of the GCB broker should be used as the value for CKPT_SERVER_HOST, not the real IP address or host name of the private node where the *condor_ckpt_server* is running.

SEC_DEFAULT_AUTHENTICATION_METHODS KERBEROS may not be used for authentication on a GCB-enabled pool. The IP addresses used in various circumstances will not be the real IP addresses of the machines. Since Kerberos stores the IP address of each host as part of the Kerberos ticket, authentication will fail on a GCB-enabled pool.

Due to the complications and security limitations that arise from running a central manager on a private node represented by GCB (both regarding the COLLECTOR_HOST and HOSTALLOW_NEGOTIATOR), we recommend that sites avoid locating a central manager on a private host whenever possible.

3.7.6 Using TCP to Send Updates to the *condor_collector*

TCP sockets are reliable, connection-based sockets that guarantee the delivery of any data sent. However, TCP sockets are fairly expensive to establish, and there is more network overhead involved in sending and receiving messages.

UDP sockets are datagrams, and are not reliable. There is very little overhead in establishing or using a UDP socket, but there is also no guarantee that the data will be delivered. Typically, the lack of guaranteed delivery for UDP does not cause problems for Condor.

Condor can be configured to use TCP sockets to send updates to the *condor_collector* instead of UDP datagrams. This feature is intended for sites where UDP updates are lost because of the underlying network. An example where this may happen is if the pool is comprised of machines across a wide area network (WAN) where UDP packets are observed to be frequently dropped.

To enable the use of TCP sockets, the following configuration setting is used:

UPDATE_COLLECTOR_WITH_TCP When set to `True`, the Condor daemons to use TCP to update the *condor_collector*, instead of the default UDP. Defaults to `False`.

When there are sufficient file descriptors, the *condor_collector* leaves established TCP sockets open, facilitating better performance. Subsequent updates can reuse an already open socket.

Each Condor daemon will have 1 socket open to the *condor_collector*. So, in a pool with N machines, each of them running a *condor_master*, *condor_schedd*, and *condor_startd*, the *condor_collector* would need at least 3*N file descriptors. If the *condor_collector* is also acting as a CCB server, it will require an additional file descriptor for each registered daemon. In typical Unix installations, the default number of file descriptors available to the *condor_collector* is only 1024. This can be modified with a configuration setting such as the following:

```
COLLECTOR_MAX_FILE_DESCRIPTORS = 1600
```

If there are not sufficient file descriptors for all of the daemons sending updates to the *condor_collector*, a warning will be printed in the *condor_collector* log file. Look for the string `file descriptor safety level exceeded`.

NOTE: At this time, `UPDATE_COLLECTOR_WITH_TCP` only affects the main *condor_collector* for the site, not any sites that a *condor_schedd* might flock to.

3.8 The Checkpoint Server

A Checkpoint Server maintains a repository for checkpoint files. Within Condor, checkpoints may be produced only for standard universe jobs. Using checkpoint servers reduces the disk requirements of submitting machines in the pool, since the submitting machines no longer need to store checkpoint files locally. Checkpoint server machines should have a large amount of disk space available, and they should have a fast connection to machines in the Condor pool.

If the spool directories are on a network file system, then checkpoint files will make two trips over the network: one between the submitting machine and the execution machine, and a second between the submitting machine and the network file server. A checkpoint server configured to use the server's local disk means that the checkpoint file will travel only once over the network, between the execution machine and the checkpoint server. The pool may also obtain checkpointing network performance benefits by using multiple checkpoint servers, as discussed below.

Note that it is a good idea to pick very stable machines for the checkpoint servers. If individual checkpoint servers crash, the Condor system will continue to operate, although poorly. While the Condor system will recover from a checkpoint server crash as best it can, there are two problems that can and will occur:

1. A checkpoint cannot be sent to a checkpoint server that is not functioning. Jobs will keep trying to contact the checkpoint server, backing off exponentially in the time they wait between attempts. Normally, jobs only have a limited time to checkpoint before they are kicked off the machine. So, if the checkpoint server is down for a long period of time, chances are that a lot of work will be lost by jobs being killed without writing a checkpoint.
2. If a checkpoint is not available from the checkpoint server, a job cannot be retrieved, and it will either have to be restarted from the beginning, or the job will wait for the server to come back on line. This behavior is controlled with the `MAX_DISCARDED_RUN_TIME` configuration variable. This variable represents the maximum amount of CPU time the job is willing to discard, by starting a job over from its beginning if the checkpoint server is not responding to requests.

3.8.1 Preparing to Install a Checkpoint Server

The location of checkpoint files changes upon the installation of a checkpoint server. A configuration change will cause currently queued jobs with checkpoints to not be able to find their checkpoints. This results in the jobs with checkpoints remaining indefinitely queued, due to the lack of finding their checkpoints. It is therefore best to either remove jobs from the queues or let them complete before installing a checkpoint server. It is advisable to shut the pool down before doing any maintenance on the checkpoint server. See section 3.10 on page 392 for details on shutting down the pool.

A graduated installation of the checkpoint server may be accomplished by configuring submit machines as their queues empty.

3.8.2 Installing the Checkpoint Server Module

The files relevant to a checkpoint server are

```
sbin/condor_ckpt_server
etc/examples/condor_config.local.ckpt.server
```

`condor_ckpt_server` is the checkpoint server binary. `condor_condor_config.local.ckpt.server` is an example configuration for a checkpoint server. The settings embodied in this file must be customized with site-specific information.

There are three steps necessary towards running a checkpoint server:

1. Configure the checkpoint server.
2. Start the checkpoint server.
3. Configure the pool to use the checkpoint server.

Configure the Checkpoint Server Place settings in the local configuration file of the checkpoint server. The file `etc/examples/condor_config.local.ckpt.server` contains a template for the needed configuration. Insert these into the local configuration file of the checkpoint server machine.

The value of `CKPT_SERVER_DIR` must be customized. This variable defines the location of checkpoint files. It is better if this location is within a very fast local file system, and preferably a RAID. The speed of this file system will have a direct impact on the speed at which checkpoint files can be retrieved from the remote machines.

The other optional variables are:

`DAEMON_LIST` Described in section 3.3.9. To have the checkpoint server managed by the *condor_master*, the `DAEMON_LIST` variable's value must list both `MASTER` and `CKPT_SERVER`. Also add `STARTD` to allow jobs to run on the checkpoint server machine. Similarly, add `SCHEDD` to permit the submission of jobs from the checkpoint server machine.

The remainder of these variables are the checkpoint server-specific versions of the Condor logging entries, as described in section 3.3.4 on page 170.

`CKPT_SERVER_LOG` The location of the checkpoint server log.

`MAX_CKPT_SERVER_LOG` Sets the maximum size of the checkpoint server log, before it is saved and the log file restarted.

`CKPT_SERVER_DEBUG` Regulates the amount of information printed in the log file. Currently, the only debug level supported is `D_ALWAYS`.

Start the Checkpoint Server To start the newly configured checkpoint server, restart Condor on that host to enable the *condor_master* to notice the new configuration. Do this by sending a *condor_restart* command from any machine with administrator access to the pool. See section 3.6.9 on page 342 for full details about IP/host-based security in Condor.

Note that when the *condor_ckpt_server* starts up, it will immediately inspect any checkpoint files in the location described by the `CKPT_SERVER_DIR` variable, and determine if any of them are stale. Stale checkpoint files will be removed.

Configure the Pool to Use the Checkpoint Server After the checkpoint server is running, modify a few configuration variables to let the other machines in the pool know about the new server:

`USE_CKPT_SERVER` A boolean value that should be set to `True` to enable the use of the checkpoint server.

`CKPT_SERVER_HOST` Provides the full host name of the machine that is now running the checkpoint server.

It is most convenient to set these variables in the pool's global configuration file, so that they affect all submission machines. However, it is permitted to configure each submission machine separately (using local configuration files), for example if it is desired that not all submission machines begin using the checkpoint server at one time. If the variable `USE_CKPT_SERVER` is set to `False`, the submission machine will not use a checkpoint server.

Once these variables are in place, send the command *condor_reconfig* to all machines in the pool, so the changes take effect. This is described in section 3.10.3 on page 395.

3.8.3 Configuring the Pool to Use Multiple Checkpoint Servers

A Condor pool may use multiple checkpoint servers. The deployment of checkpoint servers across the network improves the performance of checkpoint production. In this case, Condor machines are configured to send checkpoints to the *nearest* checkpoint server. There are two main performance benefits to deploying multiple checkpoint servers:

- Checkpoint-related network traffic is localized by intelligent placement of checkpoint servers.
- Better performance implies that jobs spend less time dealing with checkpoints, and more time doing useful work, leading to jobs having a higher success rate before returning a machine to its owner, and workstation owners see Condor jobs leave their machines quicker.

With multiple checkpoint servers running in the pool, the following configuration changes are required to make them active.

Set `USE_CKPT_SERVER` to `True` (the default) on all submitting machines where Condor jobs should use a checkpoint server. Additionally, variable `STARTER_CHOOSES_CKPT_SERVER` should be set to `True` (the default) on these submitting machines. When `True`, this variable specifies that the checkpoint server specified by the machine running the job should be used instead of the checkpoint server specified by the submitting machine. See section 3.3.8 on page 188 for more details. This allows the job to use the checkpoint server closest to the machine on which it is running, instead of the server closest to the submitting machine. For convenience, set these parameters in the global configuration file.

Second, set `CKPT_SERVER_HOST` on each machine. This identifies the full host name of the checkpoint server machine, and should be the host name of the nearest server to the machine. In the case of multiple checkpoint servers, set this in the local configuration file.

Third, send a *condor_reconfig* command to all machines in the pool, so that the changes take effect. This is described in section 3.10.3 on page 395.

After completing these three steps, the jobs in the pool will send their checkpoints to the nearest checkpoint server. On restart, a job will remember where its checkpoint was stored and retrieve it from the appropriate server. After a job successfully writes a checkpoint to a new server, it will remove any previous checkpoints left on other servers.

Note that if the configured checkpoint server is unavailable, the job will keep trying to contact that server. It will not use alternate checkpoint servers. This may change in future versions of Condor.

3.8.4 Checkpoint Server Domains

The configuration described in the previous section ensures that jobs will always write checkpoints to their nearest checkpoint server. In some circumstances, it is also useful to configure Condor to localize checkpoint read transfers, which occur when the job restarts from its last checkpoint on a new machine. To localize these transfers, it is desired to schedule the job on a machine which is near the checkpoint server on which the job's checkpoint is stored.

In terminology, all of the machines configured to use checkpoint server *A* are in *checkpoint server domain A*. To localize checkpoint transfers, jobs which run on machines in a given checkpoint server domain should continue running on machines in that domain, thereby transferring checkpoint files in a single local area of the network. There are two possible configurations which specify what a job should do when there are no available machines in its checkpoint server domain:

- The job can remain idle until a workstation in its checkpoint server domain becomes available.
- The job can try to immediately begin executing on a machine in another checkpoint server domain. In this case, the job transfers to a new checkpoint server domain.

These two configurations are described below.

The first step in implementing checkpoint server domains is to include the name of the nearest checkpoint server in the machine ClassAd, so this information can be used in job scheduling decisions. To do this, add the following configuration to each machine:

```
CkptServer = "$(CKPT_SERVER_HOST)"
STARTD_ATTRS = $(STARTD_ATTRS), CkptServer
```

For convenience, set these variables in the global configuration file. Note that this example assumes that `STARTD_ATTRS` is previously defined in the configuration. If not, then use the following configuration instead:

```
CkptServer = "$(CKPT_SERVER_HOST)"
STARTD_ATTRS = CkptServer
```

With this configuration, all machine ClassAds will include a `CkptServer` attribute, which is the name of the checkpoint server closest to this machine. So, the `CkptServer` attribute defines the checkpoint server domain of each machine.

To restrict jobs to one checkpoint server domain, modify the jobs' `Requirements` expression as follows:

```
Requirements = ((LastCkptServer == TARGET.CkptServer) || (LastCkptServer =?= UNDEFINED))
```

This `Requirements` expression uses the `LastCkptServer` attribute in the job's ClassAd, which specifies where the job last wrote a checkpoint, and the `CkptServer` attribute in the machine ClassAd, which specifies the checkpoint server domain. If the job has not yet written a checkpoint, the `LastCkptServer` attribute will be `Undefined`, and the job will be able to execute in

any checkpoint server domain. However, once the job performs a checkpoint, `LastCkptServer` will be defined and the job will be restricted to the checkpoint server domain where it started running.

To instead allow jobs to transfer to other checkpoint server domains when there are no available machines in the current checkpoint server domain, modify the jobs' `Rank` expression as follows:

```
Rank = ((LastCkptServer == TARGET.CkptServer) || (LastCkptServer =?= UNDEFINED))
```

This `Rank` expression will evaluate to 1 for machines in the job's checkpoint server domain and 0 for other machines. So, the job will prefer to run on machines in its checkpoint server domain, but if no such machines are available, the job will run in a new checkpoint server domain.

The checkpoint server domain `Requirements` or `Rank` expressions can be automatically appended to all standard universe jobs submitted in the pool using the configuration variables `APPEND_REQ_STANDARD` or `APPEND_RANK_STANDARD`. See section 3.3.14 on page 222 for more details.

3.9 DaemonCore

This section is a brief description of *DaemonCore*. DaemonCore is a library that is shared among most of the Condor daemons which provides common functionality. Currently, the following daemons use DaemonCore:

- *condor_master*
- *condor_startd*
- *condor_schedd*
- *condor_collector*
- *condor_negotiator*
- *condor_kbdd*
- *condor_quill*
- *condor_dbmsd*
- *condor_gridmanager*
- *condor_credd*
- *condor_had*
- *condor_replication*
- *condor_transferer*

- *condor_job_router*
- *condor_lease_manager*
- *condor_rooster*
- *condor_shared_port*

Most of DaemonCore's details are not interesting for administrators. However, DaemonCore does provide a uniform interface for the daemons to various Unix signals, and provides a common set of command-line options that can be used to start up each daemon.

3.9.1 DaemonCore and Unix signals

One of the most visible features that DaemonCore provides for administrators is that all daemons which use it behave the same way on certain Unix signals. The signals and the behavior DaemonCore provides are listed below:

SIGHUP Causes the daemon to reconfigure itself.

SIGTERM Causes the daemon to gracefully shutdown.

SIGQUIT Causes the daemon to quickly shutdown.

Exactly what gracefully and quickly means varies from daemon to daemon. For daemons with little or no state (the *condor_kbdd*, *condor_collector* and *condor_negotiator*) there is no difference, and both `SIGTERM` and `SIGQUIT` signals result in the daemon shutting itself down quickly. For the *condor_master*, a graceful shutdown causes the *condor_master* to ask all of its children to perform their own graceful shutdown methods. The quick shutdown causes the *condor_master* to ask all of its children to perform their own quick shutdown methods. In both cases, the *condor_master* exits after all its children have exited. In the *condor_startd*, if the machine is not claimed and running a job, both the `SIGTERM` and `SIGQUIT` signals result in an immediate exit. However, if the *condor_startd* is running a job, a graceful shutdown results in that job writing a checkpoint, while a fast shutdown does not. In the *condor_schedd*, if there are no jobs currently running, there will be no *condor_shadow* processes, and both signals result in an immediate exit. However, with jobs running, a graceful shutdown causes the *condor_schedd* to ask each *condor_shadow* to gracefully vacate the job it is serving, while a quick shutdown results in a hard kill of every *condor_shadow*, with no chance to write a checkpoint.

For all daemons, a reconfigure results in the daemon re-reading its configuration file(s), causing any settings that have changed to take effect. See section 3.3 on page 153, Configuring Condor for full details on what settings are in the configuration files and what they do.

3.9.2 DaemonCore and Command-line Arguments

The second visible feature that DaemonCore provides to administrators is a common set of command-line arguments that all daemons understand. These arguments and what they do are described below:

-a string Append a period character (`'.'`) concatenated with **string** to the file name of the log for this daemon, as specified in the configuration file.

-b Causes the daemon to start up in the background. When a DaemonCore process starts up with this option, it disassociates itself from the terminal and forks itself, so that it runs in the background. This is the default behavior for Condor daemons.

-c filename Causes the daemon to use the specified **filename** as a full path and file name as its global configuration file. This overrides the `CONDOR_CONFIG` environment variable and the regular locations that Condor checks for its configuration file which are the `condor` user's home directory and the file `/etc/condor/condor_config`.

-d Use dynamic directories. The `$(LOG)`, `$(SPOOL)`, and `$(EXECUTE)` directories are all created by the daemon at run time, and they are named by appending the parent's IP address and PID to the value in the configuration file. These values are then inherited by all children of the daemon invoked with this **-d** argument. For the *condor_master*, all Condor processes will use the new directories. If a *condor_schedd* is invoked with the *-d* argument, then only the *condor_schedd* daemon and any *condor_shadow* daemons it spawns will use the dynamic directories (named with the *condor_schedd* daemon's PID).

Note that by using a dynamically-created spool directory named by the IP address and PID, upon restarting daemons, jobs submitted to the original *condor_schedd* daemon that were stored in the old spool directory will not be noticed by the new *condor_schedd* daemon, unless you manually specify the old, dynamically-generated `SPOOL` directory path in the configuration of the new *condor_schedd* daemon.

-f Causes the daemon to start up in the foreground. Instead of forking, the daemon runs in the foreground.

NOTE: When the *condor_master* starts up daemons, it does so with the **-f** option, as it has already forked a process for the new daemon. There will be a **-f** in the argument list for all Condor daemons that the *condor_master* spawns.

-k filename For non-Windows operating systems, causes the daemon to read out a PID from the specified **filename**, and send a SIGTERM to that process. The daemon started with this optional argument waits until the daemon it is attempting to kill has exited.

-l directory Overrides the value of `LOG` as specified in the configuration files. Primarily, this option is used with the *condor_kbdd* when it needs to run as the individual user logged into the machine, instead of running as root. Regular users would not normally have permission to write files into Condor's log directory. Using this option, they can override the value of `LOG` and have the *condor_kbdd* write its log file into a directory that the user has permission to write to.

-local-name name Specify a local name for this instance of the daemon. This local name will be used to look up configuration parameters. Section 3.3.1 contains details on how this local name will be used in the configuration.

-p port Causes the daemon to bind to the specified port as its command socket. The *condor_master* daemon uses this option to ensure that the *condor_collector* and *condor_negotiator* start up using well-known ports that the rest of Condor depends upon them using.

-pidfile filename Causes the daemon to write out its PID (process id number) to the specified **filename**. This file can be used to help shutdown the daemon without first searching through the output of the Unix *ps* command.

Since daemons run with their current working directory set to the value of `LOG`, if you don't specify a full path (one that begins with a "/"), the file will be placed in the `LOG` directory.

-q Quiet output; write less verbose error messages to `stderr` when something goes wrong, and before regular logging can be initialized.

-r minutes Causes the daemon to set a timer, upon expiration of which, it sends itself a SIGTERM for graceful shutdown.

-t Causes the daemon to print out its error message to `stderr` instead of its specified log file. This option forces the **-f** option.

-v Causes the daemon to print out version information and exit.

3.10 Pool Management

Condor provides administrative tools to help with pool management. This section describes some of these tasks.

All of the commands described in this section are subject to the security policy chosen for the Condor pool. As such, the commands must be either run from a machine that has the proper authorization, or run by a user that is authorized to issue the commands. Section 3.6 on page 314 details the implementation of security in Condor.

3.10.1 Upgrading – Installing a New Version on an Existing Pool

An upgrade changes the running version of Condor from the current installation to a newer version. The safe method to install and start running a newer version of Condor in essence is: shut down the current installation of Condor, install the newer version, and then restart Condor using the newer version. To allow for falling back to the current version, place the new version in a separate directory. Copy the existing configuration files, and modify the copy to point to and use the new version, as well as incorporate any configuration variables that are new or changed in the new version. Set the

CONDOR_CONFIG environment variable to point to the new copy of the configuration, so the new version of Condor will use the new configuration when restarted.

When upgrading from a version of Condor earlier than 6.8 to more recent version, note that the configuration settings must be modified for security reasons. Specifically, the HOSTALLOW_WRITE configuration variable must be explicitly changed, or no jobs may be submitted, and error messages will be issued by Condor tools.

Another way to upgrade leaves Condor running. Condor will automatically restart itself if the *condor_master* binary is updated, and this method takes advantage of this. Download the newer version, placing it such that it does not overwrite the currently running version. With the download will be a new set of configuration files; update this new set with any specializations implemented in the currently running version of Condor. Then, modify the currently running installation by changing its configuration such that the path to binaries points instead to the new binaries. One way to do that (under Unix) is to use a symbolic link that points to the current Condor installation directory (for example, /opt/condor). Change the symbolic link to point to the new directory. If Condor is configured to locate its binaries via the symbolic link, then after the symbolic link changes, the *condor_master* daemon notices the new binaries and restarts itself. How frequently it checks is controlled by the configuration variable MASTER_CHECK_NEW_EXEC_INTERVAL , which defaults 5 minutes.

When the *condor_master* notices new binaries, it begins a graceful restart. On an execute machine, a graceful restart means that running jobs are preempted. Standard universe jobs will attempt to take a checkpoint. This could be a bottleneck if all machines in a large pool attempt to do this at the same time. If they do not complete within the cutoff time specified by the KILL policy expression (defaults to 10 minutes), then the jobs are killed without producing a checkpoint. It may be appropriate to increase this cutoff time, and a better approach may be to upgrade the pool in stages rather than all at once.

For universes other than the standard universe, jobs are preempted. If jobs have been guaranteed a certain amount of uninterrupted run time with MaxJobRetirementTime, then the job is not killed until the specified amount of retirement time has been exceeded (which is 0 by default). The first step of killing the job is a soft kill signal, which can be intercepted by the job so that it can exit gracefully, perhaps saving its state. If the job has not gone away once the KILL expression fires (10 minutes by default), then the job is forcibly hard-killed. Since the graceful shutdown of jobs may rely on shared resources such as disks where state is saved, the same reasoning applies as for the standard universe: it may be appropriate to increase the cutoff time for large pools, and a better approach may be to upgrade the pool in stages to avoid jobs running out of time.

Another time limit to be aware of is the configuration variable SHUTDOWN_GRACEFUL_TIMEOUT. This defaults to 30 minutes. If the graceful restart is not completed within this time, a fast restart ensues. This causes jobs to be hard-killed.

3.10.2 Shutting Down and Restarting a Condor Pool

Shutting Down Condor There are a variety of ways to shut down all or parts of a Condor pool. All utilize the *condor_off* tool.

To stop a single execute machine from running jobs, the *condor_off* command specifies the machine by host name.

```
condor_off -startd <hostname>
```

A running **standard** universe job will be allowed to take a checkpoint before the job is killed. A running job under another universe will be killed. If it is instead desired that the machine stops running jobs only after the currently executing job completes, the command is

```
condor_off -startd -peaceful <hostname>
```

Note that this waits indefinitely for the running job to finish, before the *condor_startd* daemon exits.

Th shut down all execution machines within the pool,

```
condor_off -all -startd
```

To wait indefinitely for each machine in the pool to finish its current Condor job, shutting down all of the execute machines as they no longer have a running job,

```
condor_off -all -startd -peaceful
```

To shut down Condor on a machine from which jobs are submitted,

```
condor_off -schedd <hostname>
```

If it is instead desired that the submit machine shuts down only after all jobs that are currently in the queue are finished, first disable new submissions to the queue by setting the configuration variable

```
MAX_JOBS_SUBMITTED = 0
```

See instructions below in section 3.10.3 for how to reconfigure a pool. After the reconfiguration, the command to wait for all jobs to complete and shut down the submission of jobs is

```
condor_off -schedd -peaceful <hostname>
```

Substitute the option **-all** for the host name, if all submit machines in the pool are to be shut down.

Restarting Condor, If Condor Daemons Are Not Running If Condor is not running, perhaps because one of the *condor_off* commands was used, then starting Condor daemons back up depends on which part of Condor is currently not running.

If no Condor daemons are running, then starting Condor is a matter of executing the *condor_master* daemon. The *condor_master* daemon will then invoke all other specified daemons on that machine. The *condor_master* daemon executes on every machine that is to run Condor.

If a specific daemon needs to be started up, and the *condor_master* daemon is already running, then issue the command on the specific machine with

```
condor_on -subsystem <subsystemname>
```

where `<subsystemname>` is replaced by the daemon's subsystem name. Or, this command might be issued from another machine in the pool (which has administrative authority) with

```
condor_on <hostname> -subsystem <subsystemname>
```

where `<subsystemname>` is replaced by the daemon's subsystem name, and `<hostname>` is replaced by the host name of the machine where this *condor_on* command is to be directed.

Restarting Condor, If Condor Daemons Are Running If Condor daemons are currently running, but need to be killed and newly invoked, the *condor_restart* tool does this. This would be the case for a new value of a configuration variable for which using *condor_reconfig* is inadequate.

To restart all daemons on all machines in the pool,

```
condor_restart -all
```

To restart all daemons on a single machine in the pool,

```
condor_restart <hostname>
```

where `<hostname>` is replaced by the host name of the machine to be restarted.

3.10.3 Reconfiguring a Condor Pool

To change a global configuration variable and have all the machines start to use the new setting, change the value within the file, and send a *condor_reconfig* command to each host. Do this with a *single* command,

```
condor_reconfig -all
```

If the global configuration file is not shared among all the machines, as it will be if using a shared file system, the change must be made to each copy of the global configuration file before issuing the *condor_reconfig* command.

Issuing a *condor_reconfig* command is inadequate for some configuration variables. For those, a restart of Condor is required. Those configuration variables that require a restart are listed in section 3.3.1 on page 158. The manual page for *condor_restart* is at 9.

3.11 The High Availability of Daemons

In the case that a key machine no longer functions, Condor can be configured such that another machine takes on the key functions. This is called *High Availability*. While high availability is generally applicable, there are currently two specialized cases for its use: when the central manager (running the *condor_negotiator* and *condor_collector* daemons) becomes unavailable, and when the machine running the *condor_schedd* daemon (maintaining the job queue) becomes unavailable.

3.11.1 High Availability of the Job Queue

For a pool where all jobs are submitted through a single machine in the pool, and there are lots of jobs, this machine becoming nonfunctional means that jobs stop running. The *condor_schedd* daemon maintains the job queue. No job queue due to having a nonfunctional machine implies that no jobs can be run. This situation is worsened by using one machine as the single submission point. For each Condor job (taken from the queue) that is executed, a *condor_shadow* process runs on the machine where submitted to handle input/output functionality. If this machine becomes nonfunctional, none of the jobs can continue. The entire pool stops running jobs.

The goal of *High Availability* in this special case is to transfer the *condor_schedd* daemon to run on another designated machine. Jobs caused to stop without finishing can be restarted from the beginning, or can continue execution using the most recent checkpoint. New jobs can enter the job queue. Without *High Availability*, the job queue would remain intact, but further progress on jobs would wait until the machine running the *condor_schedd* daemon became available (after fixing whatever caused it to become unavailable).

Condor uses its flexible configuration mechanisms to allow the transfer of the *condor_schedd* daemon from one machine to another. The configuration specifies which machines are chosen to run the *condor_schedd* daemon. To prevent multiple *condor_schedd* daemons from running at the same time, a lock (semaphore-like) is held over the job queue. This synchronizes the situation in which control is transferred to a secondary machine, and the primary machine returns to functionality. Configuration variables also determine time intervals at which the lock expires, and periods of time that pass between polling to check for expired locks.

To specify a single machine that would take over, if the machine running the *condor_schedd* daemon stops working, the following additions are made to the local configuration of any and all machines that are able to run the *condor_schedd* daemon (becoming the single pool submission

point):

```
MASTER_HA_LIST = SCHEDD
SPOOL = /share/spool
HA_LOCK_URL = file:/share/spool
VALID_SPOOL_FILES = $(VALID_SPOOL_FILES), SCHEDD.lock
```

Configuration macro MASTER_HA_LIST identifies the *condor_schedd* daemon as the daemon that is to be watched to make sure that it is running. Each machine with this configuration must have access to the lock (the job queue) which synchronizes which single machine does run the *condor_schedd* daemon. This lock and the job queue must both be located in a shared file space, and is currently specified only with a file URL. The configuration specifies the shared space (SPOOL), and the URL of the lock. *condor_preen* is not currently aware of the lock file and will delete it if it is placed in the SPOOL directory, so be sure to add SCHEDD.lock to VALID_SPOOL_FILES .

As Condor starts on machines that are configured to run the single *condor_schedd* daemon, the *condor_master* daemon of the first machine that looks at (polls) the lock and notices that no lock is held. This implies that no *condor_schedd* daemon is running. This *condor_master* daemon acquires the lock and runs the *condor_schedd* daemon. Other machines with this same capability to run the *condor_schedd* daemon look at (poll) the lock, but do not run the daemon, as the lock is held. The machine running the *condor_schedd* daemon renews the lock periodically.

If the machine running the *condor_schedd* daemon fails to renew the lock (because the machine is not functioning), the lock times out (becomes stale). The lock is released by the *condor_master* daemon if *condor_off* or *condor_off -schedd* is executed, or when the *condor_master* daemon knows that the *condor_schedd* daemon is no longer running. As other machines capable of running the *condor_schedd* daemon look at the lock (poll), one machine will be the first to notice that the lock has timed out or been released. This machine (correctly) interprets this situation as the *condor_schedd* daemon is no longer running. This machine's *condor_master* daemon then acquires the lock and runs the *condor_schedd* daemon.

See section 3.3.9, in the section on *condor_master* Configuration File Macros for details relating to the configuration variables used to set timing and polling intervals.

Working with Remote Job Submission

Remote job submission requires identification of the job queue, submitting with a command similar to:

```
% condor_submit -remote condor@example.com myjob.submit
```

This implies the identification of a single *condor_schedd* daemon, running on a single machine. With the high availability of the job queue, there are multiple *condor_schedd* daemons, of which only one at a time is acting as the single submission point. To make remote submission of jobs work properly, set the configuration variable SCHEDD_NAME in the local configuration to have the same value for each potentially running *condor_schedd* daemon. In addition, the value chosen for the

variable SCHEDD_NAME will need to include the at symbol (@), such that Condor will not modify the value set for this variable. See the description of MASTER_NAME in section 3.3.9 on page 194 for defaults and composition of valid values for SCHEDD_NAME. As an example, include in each local configuration a value similar to:

```
SCHEDD_NAME = had-schedd@
```

Then, with this sample configuration, the submit command appears as:

```
% condor_submit -remote had-schedd@  myjob.submit
```

3.11.2 High Availability of the Central Manager

Interaction with Flocking

The Condor high availability mechanisms discussed in this section currently do not work well in configurations involving flocking. The individual problems listed listed below interact to make the situation worse. Because of these problems, we advise against the use of flocking to pools with high availability mechanisms enabled.

- The *condor_schedd* has a hard configured list of *condor_collector* and *condor_negotiator* daemons, and does not query redundant collectors to get the current *condor_negotiator*, as it does when communicating with its local pool. As a result, if the default *condor_negotiator* fails, the *condor_schedd* does not learn of the failure, and thus, talk to the new *condor_negotiator*.
- When the *condor_negotiator* is unable to communicate with a *condor_collector*, it utilizes the next *condor_collector* within the list. Unfortunately, it does not start over at the top of the list. When combined with the previous problem, a backup *condor_negotiator* will never get jobs from a flocked *condor_schedd*.

Introduction

The *condor_negotiator* and *condor_collector* daemons are the heart of the Condor matchmaking system. The availability of these daemons is critical to a Condor pool's functionality. Both daemons usually run on the same machine, most often known as the central manager. The failure of a central manager machine prevents Condor from matching new jobs and allocating new resources. High availability of the *condor_negotiator* and *condor_collector* daemons eliminates this problem.

Configuration allows one of multiple machines within the pool to function as the central manager. While there are may be many active *condor_collector* daemons, only a single, active *condor_negotiator* daemon will be running. The machine with the *condor_negotiator* daemon running is the active central manager. The other potential central managers each have a *condor_collector* daemon running; these are the idle central managers.

All submit and execute machines are configured to report to all potential central manager machines.

Each potential central manager machine runs the high availability daemon, *condor_had*. These daemons communicate with each other, constantly monitoring the pool to ensure that one active central manager is available. If the active central manager machine crashes or is shut down, these daemons detect the failure, and they agree on which of the idle central managers is to become the active one. A protocol determines this.

In the case of a network partition, idle *condor_had* daemons within each partition detect (by the lack of communication) a partitioning, and then use the protocol to chose an active central manager. As long as the partition remains, and there exists an idle central manager within the partition, there will be one active central manager within each partition. When the network is repaired, the protocol returns to having one central manager.

Through configuration, a specific central manager machine may act as the primary central manager. While this machine is up and running, it functions as the central manager. After a failure of this primary central manager, another idle central manager becomes the active one. When the primary recovers, it again becomes the central manager. This is a recommended configuration, if one of the central managers is a reliable machine, which is expected to have very short periods of instability. An alternative configuration allows the promoted active central manager (in the case that the central manager fails) to stay active after the failed central manager machine returns.

This high availability mechanism operates by monitoring communication between machines. Note that there is a significant difference in communications between machines when

1. a machine is down
2. a specific daemon (the *condor_had* daemon in this case) is not running, yet the machine is functioning

The high availability mechanism distinguishes between these two, and it operates based only on first (when a central manager machine is down). A lack of executing daemons does *not* cause the protocol to choose or use a new active central manager.

The central manager machine contains state information, and this includes information about user priorities. The information is kept in a single file, and is used by the central manager machine. Should the primary central manager fail, a pool with high availability enabled would lose this information (and continue operation, but with re-initialized priorities). Therefore, the *condor_replication* daemon exists to replicate this file on all potential central manager machines. This daemon promulgates the file in a way that is safe from error, and more secure than dependence on a shared file system copy.

The *condor_replication* daemon runs on each potential central manager machine as well as on the active central manager machine. There is a unidirectional communication between the *condor_had* daemon and the *condor_replication* daemon on each machine. To properly do its job, the *condor_replication* daemon must transfer state files. When it needs to transfer a file, the *condor_replication* daemons at both the sending and receiving ends of the transfer invoke the *con-*

dor_transferer daemon. These short lived daemons do the task of file transfer and then exit. Do not place TRANSFERER into DAEMON_LIST, as it is not a daemon that the *condor_master* should invoke or watch over.

Configuration

The high availability of central manager machines is enabled through configuration. It is disabled by default. All machines in a pool must be configured appropriately in order to make the high availability mechanism work. See section 3.3.30, for definitions of these configuration variables.

The stabilization period is the time it takes for the *condor_had* daemons to detect a change in the pool state such as an active central manager failure or network partition, and recover from this change. It may be computed using the following formula:

```
stabilization period = 12 * (number of central managers) *
                       $(HAD_CONNECTION_TIMEOUT)
```

To disable the high availability of central managers mechanism, it is sufficient to remove HAD, REPLICATION, and NEGOTIATOR from the DAEMON_LIST configuration variable on all machines, leaving only one *condor_negotiator* in the pool.

To shut down a currently operating high availability mechanism, follow the given steps. All commands must be invoked from a host which has administrative permissions on all central managers. The first three commands kill all *condor_had*, *condor_replication*, and all running *condor_negotiator* daemons. The last command is invoked on the host where the single *condor_negotiator* daemon is to run.

1. `condor_off -all -neg`
2. `condor_off -all -subsystem -replication`
3. `condor_off -all -subsystem -had`
4. `condor_on -neg`

When configuring *condor_had* to control the *condor_negotiator*, if the default backoff constant value is too small, it can result in a churning of the *condor_negotiator*, especially in cases in which the primary negotiator is unable to run due to misconfiguration. In these cases, the *condor_master* will kill the *condor_had* after the *condor_negotiator* exists, wait a short period, then restart *condor_had*. The *condor_had* will then win the election, so the secondary *condor_negotiator* will be killed, and the primary will be restarted, only to exit again. If this happens too quickly, neither *condor_negotiator* will run long enough to complete a negotiation cycle, resulting in no jobs getting started. Increasing this value via MASTER_HAD_BACKOFF_CONSTANT to be larger than a typical negotiation cycle can help solve this problem.

To run a high availability pool without the replication feature, do the following operations:

1. Set the HAD_USE_REPLICATION configuration variable to False, and thus disable the replication on configuration level.
2. Remove REPLICATION from both DAEMON_LIST and DC_DAEMON_LIST in the configuration file.

Sample Configuration

This section provides sample configurations for high availability. The two parts to this are the configuration for the potential central manager machines, and the configuration for the machines within the pool that will *not* be central managers.

This is a sample configuration relating to the high availability of central managers. This is for the potential central manager machines.

```
##############################################################################
# A sample configuration file for central managers, to enable the        #
# the high availability  mechanism.                                      #
##############################################################################

# unset these two macros
NEGOTIATOR_HOST=
CONDOR_HOST=

##############################################################################
## THE FOLLOWING MUST BE IDENTICAL ON ALL POTENTIAL CENTRAL MANAGERS.   #
##############################################################################
## For simplicity in writing other expressions, define a variable
## for each potential central manager in the pool.
## These are samples.
CENTRAL_MANAGER1 = cm1.domain.name
CENTRAL_MANAGER2 = cm2.domain.name
## A list of all potential central managers in the pool.
COLLECTOR_HOST = $(CENTRAL_MANAGER1),$(CENTRAL_MANAGER2)

## Define the port number on which the condor_had daemon will
## listen.  The port must match the port number used
## for when defining HAD_LIST.  This port number is
## arbitrary; make sure that there is no port number collision
## with other applications.
HAD_PORT = 51450
HAD_ARGS = -p $(HAD_PORT)

## The following macro defines the port number condor_replication will listen
## on on this machine. This port should match the port number specified
## for that replication daemon in the REPLICATION_LIST
## Port number is arbitrary (make sure no collision with other applications)
## This is a sample port number
REPLICATION_PORT = 41450
REPLICATION_ARGS = -p $(REPLICATION_PORT)

## The following list must contain the same addresses
## as HAD_LIST. In addition, for each hostname, it should specify
```

```
## the port number of condor_replication daemon running on that host.
## This parameter is mandatory and has no default value
REPLICATION_LIST = \
$(CENTRAL_MANAGER1):$(REPLICATION_PORT), \
$(CENTRAL_MANAGER2):$(REPLICATION_PORT)

## The following list must contain the same addresses in the same order
## as COLLECTOR_HOST. In addition, for each hostname, it should specify
## the port number of condor_had daemon running on that host.
## The first machine in the list will be the PRIMARY central manager
## machine, in case HAD_USE_PRIMARY is set to true.
HAD_LIST = \
$(CENTRAL_MANAGER1):$(HAD_PORT), \
$(CENTRAL_MANAGER2):$(HAD_PORT)


## HAD connection time.
## Recommended value is 2 if the central managers are on the same subnet.
## Recommended value is 5 if Condor security is enabled.
## Recommended value is 10 if the network is very slow, or
## to reduce the sensitivity of HA daemons to network failures.
HAD_CONNECTION_TIMEOUT = 2

##If true, the first central manager in HAD_LIST is a primary.
HAD_USE_PRIMARY = true


##--------------------------------------------------------------------
##  Host/IP access levels
##--------------------------------------------------------------------

##  What machines have administrative rights for your pool?  This
##  defaults to your central manager.  You should set it to the
##  machine(s) where whoever is the condor administrator(s) works
##  (assuming you trust all the users who log into that/those
##  machine(s), since this is machine-wide access you're granting).
HOSTALLOW_ADMINISTRATOR = $(COLLECTOR_HOST)


##  Negotiator access.  Machines listed here are trusted central
##  managers.  You should normally not have to change this.
HOSTALLOW_NEGOTIATOR = $(COLLECTOR_HOST)

######################################################################
## THE PARAMETERS BELOW ARE ALLOWED TO BE DIFFERENT ON EACH       #
## CENTRAL MANAGERS                                              #
## THESE ARE MASTER SPECIFIC PARAMETERS
######################################################################

## The location of executable files
HAD         = $(SBIN)/condor_had
REPLICATION = $(SBIN)/condor_replication
TRANSFERER  = $(SBIN)/condor_transferer

## the master should start at least these five daemons
DAEMON_LIST = MASTER, COLLECTOR, NEGOTIATOR, HAD, REPLICATION
## DC_Daemon list should contain at least these five
DC_DAEMON_LIST = +HAD, REPLICATION
```

```
## Enables/disables the replication feature of HAD daemon
## Default: no
HAD_USE_REPLICATION = true

## Name of the file from the SPOOL directory that will be replicated
## Default: $(SPOOL)/Accountantnew.log
STATE_FILE = $(SPOOL)/Accountantnew.log

## Period of time between two successive awakenings of the replication daemon
## Default: 300
REPLICATION_INTERVAL = 300

# Period of time, in which transferer daemons have to accomplish the
## downloading/uploading process
## Default: 300
MAX_TRANSFERER_LIFETIME = 300

## Period of time between two successive sends of ClassAds to the collector by HAD
## Default: 300
HAD_UPDATE_INTERVAL = 300


## The HAD controls the negotiator, and should have a larger
## backoff constant
MASTER_NEGOTIATOR_CONTROLLER = HAD
MASTER_HAD_BACKOFF_CONSTANT = 360

## The size of the log file
MAX_HAD_LOG = 640000
## debug level
HAD_DEBUG = D_COMMAND
## location of the condor_had log file
HAD_LOG = $(LOG)/HADLog

## The size of replication log file
MAX_REPLICATION_LOG = 640000
## Replication debug level
REPLICATION_DEBUG = D_COMMAND
## Replication log file
REPLICATION_LOG = $(LOG)/ReplicationLog

## The size of transferer log file
MAX_TRANSFERER_LOG = 640000
## Replication debug level
TRANSFERER_DEBUG = D_COMMAND
## Replication log file
TRANSFERER_LOG = $(LOG)/TransferLog
```

Machines that are not potential central managers also require configuration. The following is a sample configuration relating to high availability for machines that will *not* be central managers.

```
##############################################################################
# Sample configuration relating to high availability for machines        #
# that DO NOT run the condor_had daemon.                                 #
##############################################################################
```

```
#unset these variables
NEGOTIATOR_HOST =
CONDOR_HOST =

## For simplicity define a variable for each potential central manager
## in the pool.
CENTRAL_MANAGER1 = cm1.cs.technion.ac.il
CENTRAL_MANAGER2 = cm2.cs.technion.ac.il
## List of all potential central managers in the pool
COLLECTOR_HOST = $(CENTRAL_MANAGER1),$(CENTRAL_MANAGER2)

##--------------------------------------------------------------------
##  Host/IP access levels
##--------------------------------------------------------------------

##  Negotiator access.  Machines listed here are trusted central
##  managers.  You should normally not need to change this.
HOSTALLOW_NEGOTIATOR = $(COLLECTOR_HOST)

##  Now, with flocking (and HA) we need to let the SCHEDD trust the other
##  negotiators we are flocking with as well.  You should normally
##  not need to change this.
HOSTALLOW_NEGOTIATOR_SCHEDD = $(COLLECTOR_HOST)
```

3.12 Quill

Quill is an optional component of Condor that maintains a mirror of Condor operational data in a relational database. The *condor_quill* daemon updates the data in the relation database, and the *condor_dbmsd* daemon maintains the database itself.

As of Condor version 7.5.5, Quill is distributed only with the source code. It is not included in the builds of Condor provided by UW, but it is available as a feature that can be enabled by those who compile Condor from the source code. Find the code within the `condor_contrib` directory, in the directories `condor_tt` and `condor_dbmsd`.

3.12.1 Installation and Configuration

Quill uses the *PostgreSQL* database management system. Quill uses the *PostgreSQL* server as its back end and client library, *libpq* to talk to the server. We **strongly recommend** the use of version 8.2 or later due to its integrated facilities of certain key database maintenance tasks, and stronger security features.

Obtain *PostgreSQL* from

http://www.postgresql.org/ftp/source/

Installation instructions are detailed in: http://www.postgresql.org/docs/8.2/static/installation.html

Configure *PostgreSQL* after installation:

1. Initialize the database with the *PostgreSQL* command `initdb`.
2. Configure to accept TCP/IP connections. For *PostgreSQL* version 8, use the `listen_addresses` variable in `postgresql.conf` file as a guide. For example, `listen_addresses = '*'` means listen on any IP interface.
3. Configure automatic vacuuming. Ensure that these variables with these defaults are commented in and/or set properly in the `postgresql.conf` configuration file:

```
# Turn on/off automatic vacuuming
autovacuum = on

# time between autovacuum runs, in secs
autovacuum_naptime = 60

# min # of tuple updates before vacuum
autovacuum_vacuum_threshold = 1000

# min # of tuple updates before analyze
autovacuum_analyze_threshold = 500

# fraction of rel size before vacuum
autovacuum_vacuum_scale_factor = 0.4

# fraction of rel size before analyze
autovacuum_analyze_scale_factor = 0.2

# default vacuum cost delay for
   # autovac, -1 means use
   # vacuum_cost_delay
autovacuum_vacuum_cost_delay = -1

# default vacuum cost limit for
   # autovac, -1 means use
   # vacuum_cost_limit
autovacuum_vacuum_cost_limit = -1
```

4. Configure *PostgreSQL* to accept TCP/IP connections from specific hosts. Modify the `pg_hba.conf` file (which usually resides in the *PostgreSQL* server's data directory). Access is required by the *condor_quill* daemon, as well as the database users "**quillreader**" and "**quillwriter**". For example, to give database users "**quillreader**" and "**quillwriter**" password-enabled access to all databases on current machine from any machine in the 128.105.0.0/16 subnet, add the following:

```
host  all  quillreader  128.105.0.0  255.255.0.0  md5
host  all  quillwriter  128.105.0.0  255.255.0.0  md5
```

   Note that in addition to the database specified by the configuration variable `QUILL_DB_NAME`, the *condor_quill* daemon also needs access to the database "tem-

plate1". In order to create the database in the first place, the *condor_quill* daemon needs to connect to the database.

5. Start the *PostgreSQL* server service. See the installation instructions for the appropriate method to start the service at http://www.postgresql.org/docs/8.2/static/installation.html

6. The *condor_quill* and *condor_dbmsd* daemons and client tools connect to the database as users "**quillreader**" and "**quillwriter**". These are database users, not operating system users. The two types of users are quite different from each other. If these database users do not exist, add them using the *createuser* command supplied with the installation. Assign them with appropriate passwords; these passwords will be used by the Quill tools to connect to the database in a secure way. User "**quillreader**" should not be allowed to create more databases nor create more users. User "**quillwriter**" should not be allowed to create more users, however it should be allowed to create more databases. The following commands create the two users with the appropriate permissions, and be ready to enter the corresponding passwords when prompted.

```
/path/to/postgreSQL/bin/directory/createuser quillreader \
--no-createdb --no-createrole --pwprompt

/path/to/postgreSQL/bin/directory/createuser quillwriter \
--createdb --no-createrole --pwprompt
```

Answer "no" to the question about the ability for role creation.

7. Create a database for Quill to store data in with the `createdb` command. Create this database with the "**quillwriter**" user as the owner. A sample command to do this is

```
createdb -O quillwriter quill
```

`quill` is the database name to use with the `QUILL_DB_NAME` configuration variable.

8. The *condor_quill* and *condor_dbmsd* daemons need read and write access to the database. They connect as user "**quillwriter**", which has owner privileges to the database. Since this gives all access to the "**quillwriter**" user, its password cannot be stored in a public place (such as in a ClassAd). For this reason, the "**quillwriter**" password is stored in a file named `.pgpass` in the Condor spool directory. Appropriate protections on this file guarantee secure access to the database. This file must be created and protected by the site administrator; if this file does not exist as and where expected, the *condor_quill* and *condor_dbmsd* daemons log an error and exit. The `.pgpass` file contains a single line that has fields separated by colons and is properly terminated by an operating system specific newline character (Unix) or CRLF (Windows). The first field may be either the machine name and fully qualified domain, or it may be a dotted quad IP address. This is followed by four fields containing: the TCP port number, the name of the database, the "quillwriter" user name, and the password. The form used in the first field must exactly match the value set for the configuration variable `QUILL_DB_IP_ADDR`. Condor uses a string comparison between the two, and it does not resolve the host names to compare IP addresses. Example:

```
machinename.cs.wisc.edu:5432:quill:quillwriter:password
```

After the *PostgreSQL* database is initialized and running, the Quill schema must be loaded into it. First, load the plsql programming language into the server:

```
createlang plpgsql [databasename]
```

Then, load the Quill schema from the sql files in the `sql` subdirectory of the Condor release directory:

```
psql [databasename] [username] < common_createddl.sql
psql [databasename] [username] < pgsql_createddl.sql
```

where `[username]` will be `quillwriter`.

After *PostgreSQL* is configured and running, Condor must also be configured to use Quill, since by default Quill is configured to be off.

Add the file `.pgpass` to the `VALID_SPOOL_FILES` variable, since *condor_preen* must be told not to delete this file. This step may not be necessary, depending on which version of Condor you are upgrading from.

Set up configuration variables that are specific to the installation, and check that the `HISTORY` variable is set.

```
HISTORY                = $(SPOOL)/history
QUILL_ENABLED          = TRUE
QUILL_USE_SQL_LOG      = FALSE
QUILL_NAME             = some-unique-quill-name.cs.wisc.edu
QUILL_DB_USER          = quillwriter
QUILL_DB_NAME          = database-for-some-unique-quill-name
QUILL_DB_IP_ADDR       = databaseIPaddress:port
# the following parameter's units is in seconds
QUILL_POLLING_PERIOD   = 10
QUILL_HISTORY_DURATION = 30
QUILL_MANAGE_VACUUM    = FALSE
QUILL_IS_REMOTELY_QUERYABLE = TRUE
QUILL_DB_QUERY_PASSWORD =  password-for-database-user-quillreader
QUILL_ADDRESS_FILE     = $(LOG)/.quill_address
QUILL_DB_TYPE          = PGSQL
# The Purge and Reindex intervals are in seconds
DATABASE_PURGE_INTERVAL = 86400
DATABASE_REINDEX_INTERVAL = 86400
# The History durations are all in days
QUILL_RESOURCE_HISTORY_DURATION  = 7
QUILL_RUN_HISTORY_DURATION = 7
QUILL_JOB_HISTORY_DURATION = 3650
#The DB Size limit is in gigabytes
QUILL_DBSIZE_LIMIT     = 20
QUILL_MAINTAIN_DB_CONN  = TRUE
SCHEDD_SQLLOG          = $(LOG)/schedd_sql.log
SCHEDD_DAEMON_AD_FILE  = $(LOG)/.schedd_classad
```

The default Condor configuration file should already contain definitions for `QUILL` and `QUILL_LOG`. When upgrading from a previous version that did not have Quill to a new one that does, define these two configuration variables.

Only one machine should run the *condor_dbmsd* daemon. On this machine, add it to the DAEMON_LIST configuration variable. All Quill-enabled machines should also run the *condor_quill* daemon. The machine running the *condor_dbmsd* daemon can also run a *condor_quill* daemon. An example DAEMON_LIST for a machine running both daemons, and acting as both a submit machine and a central manager might look like the following:

```
DAEMON_LIST  = MASTER, SCHEDD, COLLECTOR, NEGOTIATOR, DBMSD, QUILL
```

The *condor_dbmsd* daemon will need configuration file entries common to all daemons. If not already in the configuration file, add the following entries:

```
DBMSD = $(SBIN)/condor_dbmsd
DBMSD_ARGS = -f
DBMSD_LOG = $(LOG)/DbmsdLog
MAX_DBMSD_LOG = 10000000
```

Descriptions of these and other configuration variables are in section 3.3.31. Here are further brief details:

QUILL_DB_NAME and QUILL_DB_IP_ADDR These two variables are used to determine the location of the database server that this Quill would talk to, and the name of the database that it creates. More than one Quill server can talk to the same database server. This can be accomplished by letting all the QUILL_DB_IP_ADDR values point to the same database server.

QUILL_DB_USER This is the *PostgreSQL* user that Quill will connect as to the database. We recommend "**quillwriter**" for this setting. There is no default setting for QUILL_DB_USER, so it must be specified in the configuration file.

QUILL_NAME Each *condor_quill* daemon in the pool has to be uniquely named.

QUILL_POLLING_PERIOD This controls the frequency with which Quill polls the job_queue.log file. By default, it is 10 seconds. Since Quill works by periodically sniffing the log file for updates and then sending those updates to the database, this variable controls the trade off between the currency of query results and Quill's load on the system, which is usually negligible.

QUILL_RESOURCE_HISTORY_DURATION, QUILL_RUN_HISTORY_DURATION, and QUILL_JOB_HISTORY_DURA These three variables control the deletion of historical information from the database. QUILL_RESOURCE_HISTORY_DURATION is the number of days historical information about the state of a resource will be kept in the database. The default for resource history is 7 days. An example of a resource is the ClassAd for a compute slot. QUILL_RUN_HISTORY_DURATION is the number of days after completion that auxiliary information about a given job will stay in the database. This includes user log events, file transfers performed by the job, the matches that were made for a job, et cetera. The default

for run history is 7 days. QUILL_JOB_HISTORY_DURATION is the number of days after completion that a given job will stay in the database. A more precise definition is the number of days since the history ad got into the history database; those two might be different, if a job is completed but stays in the queue for a while. The default for job history is 3,650 days (about 10 years.)

DATABASE_PURGE_INTERVAL As scanning the entire database for old jobs can be expensive, the other variable DATABASE_PURGE_INTERVAL is the number of seconds between two successive scans. DATABASE_PURGE_INTERVAL is set to 86400 seconds, or one day.

DATABASE_REINDEX_INTERVAL *PostgreSQL* does not aggressively maintain the index structures for deleted tuples. This can lead to bloated index structures. Quill can periodically reindex the database, which is controlled by the variable DATABASE_REINDEX_INTERVAL. DATABASE_PURGE_INTERVAL is set to 86400 seconds, or one day.

QUILL_DBSIZE_LIMIT Quill can estimate the size of the database, and send email to the Condor administrator if the database size exceeds this threshold. The estimate is checked after every DATABASE_PURGE_INTERVAL. The limit is given as gigabytes, and the default is 20.

QUILL_MAINTAIN_DB_CONN Quill can maintain an open connection the database server, which speeds up updates to the database. However, each open connection consumes resources at the database server. The default is TRUE, but for large pools we recommend setting this FALSE.

QUILL_MANAGE_VACUUM Set to False by default, this variable determines whether Quill is to perform vacuuming tasks on its tables or not. Vacuuming is a maintenance task that needs to be performed on tables in *PostgreSQL*. The frequency with which a table is vacuumed typically depends on the number of updates (inserts/deletes) performed on the table. Fortunately, with *PostgreSQL* version 8.1, vacuuming tasks can be configured to be performed automatically by the database server. We recommend that users upgrade to 8.1 and use the integrated vacuuming facilities of the database server, instead of having Quill do them. If the user does prefer having Quill perform those vacuuming tasks, it can be achieved by setting this variable to ExprTrue. However, it cannot be overstated that Quill's vacuuming policy is quite rudimentary as compared to the integrated facilities of the database server, and under high update workloads, can prove to be a bottleneck on the Quill daemon. As such, setting this variable to ExprTrue results in some warning messages in the log file regarding this issue.

QUILL_IS_REMOTELY_QUERYABLE Thanks to *PostgreSQL*, one can now remotely query both the job queue and the history tables. This variable controls whether this remote querying feature should be enabled. By default it is True. Note that even if this is False, one can still query the job queue at the remote *condor_schedd* daemon. This variable only controls whether the database tables are remotely queryable.

QUILL_DB_QUERY_PASSWORD In order for the query tools to connect to a database, they need to provide the password that is assigned to the database user "**quillreader**". This variable is then advertised by the *condor_quill* daemon to the *condor_collector*. This facility enables remote querying: remote *condor_q* query tools first ask the *condor_collector* for the password associated with a particular Quill database, and then query that database. Users who do not

have access to the *condor_collector* cannot view the password, and as such cannot query the database. Again, this password only provides read access to the database.

QUILL_ADDRESS_FILE When Quill starts up, it can place its address (IP and port) into a file. This way, tools running on the local machine do not need to query the central manager to find Quill. This feature can be turned off by commenting out the variable.

3.12.2 Four Usage Examples

1. Query a remote Quill daemon on `regular.cs.wisc.edu` for all the jobs in the queue

   ```
   condor_q -name quill@regular.cs.wisc.edu
   condor_q -name schedd@regular.cs.wisc.edu
   ```

   There are two ways to get to a Quill daemon: directly using its name as specified in the `QUILL_NAME` configuration variable, or indirectly by querying the *condor_schedd* daemon using its name. In the latter case, *condor_q* will detect if that *condor_schedd* daemon is being serviced by a database, and if so, directly query it. In both cases, the IP address and port of the database server hosting the data of this particular remote Quill daemon can be figured out by the `QUILL_DB_IP_ADDR` and `QUILL_DB_NAME` variables specified in the `QUILL_AD` sent by the quill daemon to the collector and in the `SCHEDD_AD` sent by the *condor_schedd* daemon.

2. Query a remote Quill daemon on `regular.cs.wisc.edu` for all historical jobs belonging to owner einstein.

   ```
   condor_history -name quill@regular.cs.wisc.edu einstein
   ```

3. Query the local Quill daemon for the average time spent in the queue for all non-completed jobs.

   ```
   condor_q -avgqueuetime
   ```

   The average queue time is defined as the average of `(currenttime - jobsubmissiontime)` over all jobs which are neither completed `(JobStatus == 4)` or removed `(JobStatus == 3)`.

4. Query the local Quill daemon for all historical jobs completed since Apr 1, 2005 at 13h 00m.

   ```
   condor_history -completedsince '04/01/2005 13:00'
   ```

   It fetches all jobs which got into the 'Completed' state on or after the specified time stamp. It use the *PostgreSQL* date/time syntax rules, as it encompasses most format options. See http://www.postgresql.org/docs/8.2/static/datatype-datetime.html for the various time stamp formats.

3.12.3 Quill and Security

There are several layers of security in Quill, some provided by Condor and others provided by the database. First, all accesses to the database are password-protected.

1. The query tools, *condor_q* and *condor_history* connect to the database as user "**quillreader**". The password for this user can vary from one database to another and as such, each Quill daemon advertises this password to the collector. The query tools then obtain this password from the collector and connect successfully to the database. Access to the database by the "**quillreader**" user is read-only, as this is sufficient for the query tools. The *condor_quill* daemon ensures this protected access using the sql GRANT command when it first creates the tables in the database. Note that access to the "**quillreader**" password itself can be blocked by blocking access to the collector, a feature already supported in Condor.
2. The *condor_quill* and *condor_dbmsd* daemons, on the other hand, need read and write access to the database. As such, they connect as user "**quillwriter**", who has owner privileges to the database. Since this gives all access to the "**quillwriter**" user, this password cannot be stored in a public place (such as the collector). For this reason, the "**quillwriter**" password is stored in a file called `.pgpass` in the Condor spool directory. Appropriate protections on this file guarantee secure access to the database. This file must be created and protected by the site administrator; if this file does not exist as and where expected, the *condor_quill* daemon logs an error and exits.
3. The `IsRemotelyQueryable` attribute in the Quill ClassAd advertised by the Quill daemon to the collector can be used by site administrators to disallow the database from being read by all remote Condor query tools.

3.12.4 Quill and Its RDBMS Schema

Notes:

- The type "timestamp(*precision*) with timezone" is abbreviated "ts(*precision*) w tz."
- The column O. Type is an abbreviation for Oracle Type.
- The column P. Type is an abbreviation for PostgreSQL Type.

Although the current version of Condor does not support Oracle, we anticipate supporting it in the future, so Oracle support in this schema document is for future reference.

Administrative Tables

Attributes of currencies Table			
Name	**O. Type**	**P. Type**	**Description**
datasource	varchar(4000)	varchar(4000)	Identifier of the data source.
lastupdate	ts(3) w tz	ts(3) w tz	Time of the last update sent to the database from the data source.

Attributes of error_sqllogs Table			
Name	**O. Type**	**P. Type**	**Description**
logname	varchar(100)	varchar(100)	Name of the SQL log file causing a SQL error.
host	varchar(50)	varchar(50)	The host where the SQL log resides.
lastmodified	ts(3) w tz	ts(3) w tz	The last modified time of the SQL log.
errorsql	varchar(4000)	text	The SQL statement causing an error.
logbody	clob	text	The body of the SQL log.
errormessage	varchar(4000)	varchar(4000)	The description of the error.
INDEX: Index named error_sqllog_idx on (logname, host, lastmodified)			

Attributes of maintenance_log Table			
Name	**O. Type**	**P. Type**	**Description**
eventts	ts(3) w tz	ts(3) w tz	Time the event occurred.
eventmsg	varchar(4000)	varchar(4000)	Message describing the event.

Attributes of quilldbmonitor Table			
Name	**O. Type**	**P. Type**	**Description**
dbsize	integer	integer	Size of the database in megabytes.

Attributes of quill_schema_version Table			
Name	**O. Type**	**P. Type**	**Description**
major	int	int	Major version number.
minor	int	int	Minor version number.
back_to_major	int	int	The major number of the old version this version is compatible to.
back_to_minor	int	int	The minor number of the old version this version is compatible to.

Attributes of throwns Table			
Name	**O. Type**	**P. Type**	**Description**
filename	varchar(4000)	varchar(4000)	The name of the log that was truncated.
machine_id	varchar(4000)	varchar(4000)	The machine where the truncated log resides.
log_size	numeric(38)	numeric(38)	The size of the truncated log.
throwtime	ts(3) w tz	ts(3) w tz	The time when the truncation occurred.

Daemon Tables

Attributes of daemons_horizontal Table			
Name	**O. Type**	**P. Type**	**Description**
mytype	varchar(100)	varchar(100)	The type of daemon ClassAd, e.g. "Master"
name	varchar(500)	varchar(500)	The name identifier of the daemon ClassAd.
lastreportedtime	ts(3) w tz	ts(3) w tz	Time when the daemon last reported to Quill.
monitorselftime	ts(3) w tz	ts(3) w tz	The time when the daemon last collected information about itself.
monitorselfcpuusage	numeric(38)	numeric(38)	The amount of CPU this daemon has used.
monitorselfimagesize	numeric(38)	numeric(38)	The amount of virtual memory this daemon has used.
monitorselfresidentsetsize	numeric(38)	numeric(38)	The amount of physical memory this daemon has used.
monitorselfage	integer	integer	How long the daemon has been running.
updatesequencenumber	integer	integer	The sequence number associated with the update.
updatestotal	integer	integer	The number of updates received from the daemon.
updatessequenced	integer	integer	The number of updates that were in order.
updateslost	integer	integer	The number of updates that were lost.
updateshistory	varchar(4000)	varchar(4000)	Bitmask of the last 32 updates.
lastreportedtime_epoch	integer	integer	The equivalent epoch time of last heard from.
PRIMARY KEY: (mytype, name)			
NOT NULL: mytype and name cannot be null			

Attributes of daemons_horizontal_history Table			
Name	**O. Type**	**P. Type**	**Description**
mytype	varchar(100)	varchar(100)	The type of daemon ClassAd, e.g. “Master”
name	varchar(500)	varchar(500)	The name identifier of the daemon ClassAd.
lastreportedtime	ts(3) w tz	ts(3) w tz	Time when the daemon last reported to Quill.
monitorselftime	ts(3) w tz	ts(3) w tz	The time when the daemon last collected information about itself.
monitorselfcpuusage	numeric(38)	numeric(38)	The amount of CPU this daemon has used.
monitorselfimagesize	numeric(38)	numeric(38)	The amount of virtual memory this daemon has used.
monitorselfresidentsetsize	numeric(38)	numeric(38)	The amount of physical memory this daemon has used.
monitorselfage	integer	integer	How long the daemon has been running.
updatesequencenumber	integer	integer	The sequence number associated with the update.
updatestotal	integer	integer	The number of updates received from the daemon.
updatessequenced	integer	integer	The number of updates that were in order.
updateslost	integer	integer	The number of updates that were lost.
updateshistory	varchar(4000)	varchar(4000)	Bitmask of the last 32 updates.
endtime	ts(3) w tz	ts(3) w tz	End of when the ClassAd is valid.

Attributes of daemons_vertical Table			
Name	**O. Type**	**P. Type**	**Description**
mytype	varchar(100)	varchar(100)	The type of daemon ClassAd, e.g. “Master”
name	varchar(500)	varchar(500)	The name identifier of the daemon ClassAd.
attr	varchar(4000)	varchar(4000)	Attribute name.
val	clob	text	Attribute value.
lastreportedtime	ts(3) w tz	ts(3) w tz	Time when the daemon last reported to Quill.
PRIMARY KEY: (mytype, name, attr)			
NOT NULL: mytype, name, and attr cannot be null			

Attributes of daemons_vertical_history Table			
Name	**O. Type**	**P. Type**	**Description**
mytype	varchar(100)	varchar(100)	The type of daemon ClassAd, e.g. “Master”
name	varchar(500)	varchar(500)	The name identifier of the daemon ClassAd.
lastreportedtime	ts(3) w tz	ts(3) w tz	Time when the daemon last reported to Quill.
attr	varchar(4000)	varchar(4000)	Attribute name.
val	clob	text	Attribute value.
endtime	ts(3) w tz	ts(3) w tz	End of when the ClassAd is valid.

Attributes of submitters_horizontal table			
Name	**O. Type**	**P. Type**	**Description**
name	varchar(500)	varchar(500)	Name of the submitter ClassAd.
scheddname	varchar(4000)	varchar(4000)	Name of the schedd where the submitter ad is from.
lastreportedtime	ts(3) w tz	ts(3) w tz	Last time a submitter ClassAd was sent to Quill.
idlejobs	integer	integer	Number of idle jobs of the submitter.
runningjobs	integer	integer	Number of running jobs of the submitter.
heldjobs	integer	integer	Number of held jobs of the submitter.
flockedjobs	integer	integer	Number of flocked jobs of the submitter.

Attributes of submitters_horizontal_history table			
Name	**O. Type**	**P. Type**	**Description**
name	varchar(500)	varchar(500)	Name of the submitter ClassAd.
scheddname	varchar(4000)	varchar(4000)	Name of the schedd where the submitter ad is from.
lastreportedtime	ts(3) w tz	ts(3) w tz	Last time a submitter ClassAd was sent to Quill.
idlejobs	integer	integer	Number of idle jobs of the submitter.
runningjobs	integer	integer	Number of running jobs of the submitter.
heldjobs	integer	integer	Number of held jobs of the submitter.
flockedjobs	integer	integer	Number of flocked jobs of the submitter.
endtime	ts(3) w tz	ts(3) w tz	End of when the ClassAd is valid.

Files Tables

Attributes of files Table			
Name	**O. Type**	**P. Type**	**Description**
file_id	int	int	Unique numeric identifier of the file.
name	varchar(4000)	varchar(4000)	File name.
host	varchar(4000)	varchar(4000)	Name of machine where the file is located.
path	varchar(4000)	varchar(4000)	Directory path to the file.
acl_id	integer	integer	Not yet used, null.
lastmodified	ts(3) w tz	ts(3) w tz	Timestamp of the file.
filesize	numeric(38)	numeric(38)	Size of the file in bytes.
checksum	varchar(32)	varchar(32)	MD5 checksum of the file.
PRIMARY KEY: file_id			
NOT NULL: file_id cannot be null			

Attributes of fileusages Table			
Name	**O. Type**	**P. Type**	**Description**
globaljobid	varchar(4000)	varchar(4000)	Global identifier of the job that used the file.
file_id	int	int	Numeric identifier of the file.
usagetype	varchar(4000)	varchar(4000)	Type of use of the file by the job, e.g., input, output, command.
REFERENCE: file_id references files(file_id)			

Attributes of transfers Table			
Name	**O. Type**	**P. Type**	**Description**
globaljobid	varchar(4000)	varchar(4000)	Unique global identifier for the job.
src_name	varchar(4000)	varchar(4000)	Name of the file on the source machine.
src_host	varchar(4000)	varchar(4000)	Name of the source machine.
src_port	integer	integer	Source port number used for the transfer.
src_path	varchar(4000)	varchar(4000)	Path to the file on the source machine.
src_daemon	varchar(30)	varchar(30)	Condor demon performing the transfer on the source machine.
src_protocol	varchar(30)	varchar(30)	The protocol used on the source machine.
src_credential_id	integer	integer	Not yet used, null.
src_acl_id	integer	integer	Not yet used, null.
dst_name	varchar(4000)	varchar(4000)	Name of the file on the destination machine.
dst_host	varchar(4000)	varchar(4000)	Name of the destination machine.
dst_port	integer	integer	Destination port number used for the transfer.
dst_path	varchar(4000)	varchar(4000)	Path to the file on the destination machine.
dst_daemon	varchar(30)	varchar(30)	Condor daemon receiving the transfer on the destination machine.
dst_protocol	varchar(30)	varchar(30)	The protocol used on the destination machine.
dst_credential_id	integer	integer	Not yet used, null.
dst_acl_id	integer	integer	Not yet used, null.
transfer_intermediary_id	integer	integer	Not yet used, null; will use someday if a proxy is used.
transfer_size_bytes	numeric(38)	numeric (38)	Size of the data transfered in bytes.
elapsed	numeric(38)	numeric(38)	Number of seconds that elapsed during the transfer.
checksum	varchar(256)	varchar(256)	Checksum of the file.
transfer_time	ts(3) w tz	ts(3) w tz	Time when the transfer took place.
last_modified	ts(3) w tz	ts(3) w tz	Last modified time for the file that was transfered.
is_encrypted	varchar(5)	varchar(5)	(boolean) True if the file is encrypted.
delegation_method_id	integer	integer	Not yet used, null.
completion_code	integer	integer	Indicates whether the transfer failed or succeeded.

Interface Tables

Attributes of cdb_users Table			
Name	**O. Type**	**P. Type**	**Description**
userid	varchar(30)	varchar(30)	Unique identifier of the user
password	character(32)	character(32)	Encrypted password
admin	varchar(5)	varchar(5)	(boolean) True if the user has administrator privileges

Attributes of l_eventtype Table			
Name	**O. Type**	**P. Type**	**Description**
eventtype	integer	integer	Numeric type code of the event.
description	varchar(4000)	varchar(4000)	Description of the type of event associated with the event-type code.

Attributes of l_jobstatus Table			
Name	**O. Type**	**P. Type**	**Description**
jobstatus	integer	integer	Numeric code for job status.
abbrev	char(1)	char(1)	Single letter code for job status.
description	varchar(4000)	varchar(4000)	Description of job status.
PRIMARY KEY: jobstatus			
NOT NULL: jobstatus cannot be null			

Jobs Tables

Attributes of clusterads_horizontal Table			
Name	**O. Type**	**P. Type**	**Description**
scheddname	varchar(4000)	varchar(4000)	Name of the schedd the job is submitted to.
cluster_id	integer	integer	Cluster identifier for the job.
owner	varchar(30)	varchar(30)	User who submitted the job.
jobstatus	integer	integer	Current status of the job.
jobprio	integer	integer	Priority for this job.
imagesize	numeric(38)	numeric(38)	Estimate of memory image size of the job in kilobytes.
qdate	ts(3) w tz	ts(3) w tz	Time the job was submitted to the job queue.
remoteusercpu	numeric(38)	numeric(38)	Total number of seconds of user CPU time the job used on remote machines.
remotewallclocktime	numeric(38)	numeric(38)	Committed cumulative number of seconds the job has been allocated to a machine.
cmd	clob	text	Path to and filename of the job to be executed.
args	clob	text	Arguments passed to the job.
jobuniverse	integer	integer	The Condor universe used by the job.
PRIMARY KEY: (scheddname, cluster_id)			
NOT NULL: scheddname and cluster_id cannot be null			

Attributes of clusterads_vertical Table			
Name	**O. Type**	**P. Type**	**Description**
scheddname	varchar(4000)	varchar(4000)	Name of the schedd that the job is submitted to.
cluster_id	integer	integer	Cluster identifier for the job.
attr	varchar(2000)	varchar(2000)	Attribute name.
val	clob	text	Attribute value.
PRIMARY KEY: (scheddname, cluster_id, attr)			

Attributes of jobs_horizontal_history Table – Part 1 of 3			
Name	**O. Type**	**P. Type**	**Description**
scheddname	varchar(4000)	varchar(4000)	Name of the schedd that submitted the job.
scheddbirthdate	integer	integer	The birth date of the schedd where the job is submitted.
cluster_id	integer	integer	Cluster identifier for the job.
proc_id	integer	integer	Process identifier for the job.
qdate	ts(3) w tz	ts(3) w tz	Time the job was submitted to the job queue.
owner	varchar(30)	varchar(30)	User who submitted the job.
globaljobid	varchar(4000)	varchar(4000)	Unique global identifier for the job.
numckpts	integer	integer	Number of checkpoints written by the job during its lifetime.
numrestarts	integer	integer	Number of restarts from a checkpoint attempted by the job in its lifetime.
numsystemholds	integer	integer	Number of times Condor-G placed the job on hold.
condorversion	varchar(4000)	varchar(4000)	Version of Condor that ran the job.
condorplatform	varchar(4000)	varchar(4000)	Platform of the computer where the schedd runs.
rootdir	varchar(4000)	varchar(4000)	Root directory on the system where the job is submitted from.
iwd	varchar(4000)	varchar(4000)	Initial working directory of the job.
jobuniverse	integer	integer	The Condor universe used by the job.
cmd	clob	text	Path to and filename of the job to be executed.
minhosts	integer	integer	Minimum number of hosts that must be in the claimed state for this job, before the job may enter the running state.
maxhosts	integer	integer	Maximum number of hosts this job would like to claim.
jobprio	integer	integer	Priority for this job.
negotiation_user_name	varchar(4000)	varchar(4000)	User name in which the job is negotiated.
env	clob	text	Environment under which the job ran.
userlog	varchar(4000)	varchar(4000)	User log where the job events are written to.
coresize	numeric(38)	numeric(38)	Maximum allowed size of the core file.
Table Continues on Next Page			

Attributes of jobs_horizontal_history Table – Part 2 of 3			
Name	**O. Type**	**P. Type**	**Description**
killsig	varchar(4000)	varchar(4000)	Signal to be sent if the job is put on hold.
stdin	varchar(4000)	varchar(4000)	The file used as stdin.
transferin	varchar(5)	varchar(5)	(boolean) For globus universe jobs. True if input should be transferred to the remote machine.
stdout	varchar(4000)	varchar(4000)	The file used as stdout.
transferout	varchar(5)	varchar(5)	(boolean) For globus universe jobs. True if output should be transferred back to the submit machine.
stderr	varchar(4000)	varchar(4000)	The file used as stderr.
transfererr	varchar(5)	varchar(5)	(boolean) For globus universe jobs. True if error output should be transferred back to the submit machine.
shouldtransferfiles	varchar (4000)	varchar(4000)	Whether Condor should transfer files to and from the machine where the job runs.
transferfiles	varchar(4000)	varchar(4000)	Depreciated. Similar to shouldtransferfiles.
executablesize	numeric(38)	numeric(38)	Size of the executable in kilobytes.
diskusage	integer	integer	Size of the executable and input files to be transferred.
filesystemdomain	varchar(4000)	varchar(4000)	Name of the networked file system used by the job.
args	clob	text	Arguments passed to the job.
lastmatchtime	ts(3) w tz	ts(3) w tz	Time when the job was last successfully matched with a resource.
numjobmatches	integer	integer	Number of times the negotiator matches the job with a resource.
jobstartdate	ts(3) w tz	ts(3) w tz	Time when the job first began running.
jobcurrentstartdate	ts(3) w tz	ts(3) w tz	Time when the job's current run started.
jobruncount	integer	integer	Number of times a shadow has been started for the job.
filereadcount	numeric(38)	numeric(38)	Number of read(2) calls the job made (only standard universe).
filereadbytes	numeric(38)	numeric(38)	Number of bytes read by the job (only standard universe).
filewritecount	numeric(38)	numeric(38)	Number of write calls the job made (only standard universe).
filewritebytes	numeric(38)	numeric(38)	Number of bytes written by the job (only standard universe).

Table Continues on Next Page

Attributes of jobs_horizontal_history Table – Part 3 of 3			
Name	**O. Type**	**P. Type**	**Description**
fileseekcount	numeric(38)	numeric(38)	Number of seek calls that this job made (only standard universe).
totalsuspensions	integer	integer	Number of times the job has been suspended during its lifetime
imagesize	numeric(38)	numeric(38)	Estimate of memory image size of the job in kilobytes.
exitstatus	integer	integer	No longer used by Condor.
localusercpu	numeric(38)	numeric(38)	Number of seconds of user CPU time the job used on the submit machine.
localsyscpu	numeric(38)	numeric(38)	Number of seconds of system CPU time the job used on the submit machine.
remoteusercpu	numeric(38)	numeric(38)	Number of seconds of user CPU time the job used on remote machines.
remotesyscpu	numeric(38)	numeric(38)	Number of seconds of system CPU time the job used on remote machines.
bytessent	numeric(38)	numeric(38)	Number of bytes sent to the job.
bytesrecvd	numeric(38)	numeric(38)	Number of bytes received by the job.
rscbytessent	numeric(38)	numeric(38)	Number of remote system call bytes sent to the job.
rscbytesrecvd	numeric(38)	numeric(38)	Number of remote system call bytes received by the job.
exitcode	integer	integer	Exit return code of the user job. Used when a job exits by means other than a signal.
jobstatus	integer	integer	Current status of the job.
enteredcurrentstatus	ts(3) w tz	ts(3) w tz	Time the job entered into its current status.
remotewallclocktime	numeric(38)	numeric(38)	Cumulative number of seconds the job has been allocated to a machine.
lastremotehost	varchar(4000)	varchar(4000)	The remote host for the last run of the job.
completiondate	ts(3) w tz	ts(3) w tz	Time when the job completed; 0 if job has not yet completed.
enteredhistorytable	ts(3) w tz	ts(3) w tz	Time when the job entered the history table.
PRIMARY KEY: (scheddname, scheddbirthdate, cluster_id, proc_id)			
NOT NULL: scheddname, scheddbirthdate, cluster_id, and proc_id cannot be null			
INDEX: Index named hist_h_i_owner on owner			

Attributes of jobs_vertical_history Table			
Name	**O. Type**	**P. Type**	**Description**
scheddname	varchar(4000)	varchar(4000)	Name of the schedd that submitted the job.
scheddbirthdate	integer	integer	The birth date of the schedd where the job is submitted.
cluster_id	integer	integer	Cluster identifier for the job.
proc_id	integer	integer	Process identifier for the job.
attr	varchar(2000)	varchar(2000)	Attribute name.
val	clob	text	Attribute value.
PRIMARY KEY: (scheddname, scheddbirthdate, cluster_id, proc_id, attr)			
NOT NULL: scheddname, scheddbirthdate, cluster_id, proc_id, and attr cannot be null			

Attributes of procads_horizontal Table			
Name	**O. Type**	**P. Type**	**Description**
scheddname	varchar(4000)	varchar(4000)	Name of the schedd that submitted the job.
cluster_id	integer	integer	Cluster identifier for the job.
proc_id	integer	integer	Process identifier for the job.
jobstatus	integer	integer	Current status of the job.
imagesize	numeric(38)	numeric(38)	Estimate of memory image size of the job in kilobytes.
remoteusercpu	numeric(38)	numeric(38)	Total number of seconds of user CPU time the job used on remote machines.
remotewallclocktime	numeric(38)	numeric(38)	Cumulative number of seconds the job has been allocated to a machine.
remotehost	varchar(4000)	varchar(4000)	Name of the machine running the job.
globaljobid	varchar(4000)	varchar(4000)	Unique global identifier for the job.
jobprio	integer	integer	Priority of the job.
args	clob	text	Arguments passed to the job.
shadowbday	ts(3) w tz	ts(3) w tz	The time when the shadow was started.
enteredcurrentstatus	ts(3) w tz	ts(3) w tz	Time the job entered its current status.
numrestarts	integer	integer	Number of times the job has restarted.
PRIMARY KEY: (scheddname, cluster_id, proc_id)			
NOT NULL: scheddname, cluster_id, and proc_id cannot be null			

Attributes of procads_vertical Table			
Name	**O. Type**	**P. Type**	**Description**
scheddname	varchar(4000)	varchar(4000)	Name of the schedd that submitted the job.
cluster_id	integer	integer	Cluster identifier for the job.
proc_id	integer	integer	Process identifier for the job.
attr	varchar(2000)	varchar(2000)	Attribute name.
val	clob	text	Attribute value.

Machines Tables

Attributes of machines_horizontal Table – Part 1 of 2			
Name	**O. Type**	**P. Type**	**Description**
machine_id	varchar(4000)	varchar(4000)	Unique identifier of the machine.
opsys	varchar(4000)	varchar(4000)	Operating system running on the machine.
arch	varchar(4000)	varchar(4000)	Architecture of the machine.
state	varchar(4000)	varchar(4000)	Condor state of the machine.
activity	varchar(4000)	varchar(4000)	Condor job activity on the machine.
keyboardidle	integer	integer	Number of seconds since activity has been detected on any keyboard or mouse associated with the machine.
consoleidle	integer	integer	Number of seconds since activity has been detected on the console keyboard or mouse.
loadavg	real	real	Current load average of the machine.
condorloadavg	real	real	Portion of load average generated by Condor
totalloadavg	real	real	
virtualmemory	integer	integer	Amount of currently available virtual memory in kilobytes.
memory	integer	integer	Amount of RAM in megabytes.
totalvirtualmemory	integer	integer	
cpubusytime	integer	integer	Time in seconds since cpuisbusy became true.
cpuisbusy	varchar(5)	varchar(5)	(boolean) True when the CPU is busy.
currentrank	real	real	The machine owner's affinity for running the Condor job which it is currently hosting.
clockmin	integer	integer	Number of minutes passed since midnight.
clockday	integer	integer	The day of the week.
lastreportedtime	ts(3) w tz	ts(3) w tz	Time when the Condor central manager last received a status update from this machine.
enteredcurrentactivity	ts(3) w tz	ts(3) w tz	Time when the machine entered the current activity.
enteredcurrentstate	ts(3) w tz	ts(3) w tz	Time when the machine entered the current state.
updatesequencenumber	integer	integer	Each update includes a sequence number.
Table Continues on Next Page			

Attributes of machines_horizontal Table – Part 2 of 2			
updatestotal	integer	integer	The number of updates received from the daemon.
updatessequenced	integer	integer	The number of updates that were in order.
updateslost	integer	integer	The number of updates that were lost.
globaljobid	varchar(4000)	varchar(4000)	Unique global identifier for the job.
lastreportedtime_epoch	integer	integer	The equivalent epoch time of lastreported-time.
PRIMARY KEY: machine_id			

Attributes of machines_horizontal_history Table – Part 1 of 2			
Name	**O. Type**	**P. Type**	**Description**
machine_id	varchar(4000)	varchar(4000)	Unique identifier of the machine.
opsys	varchar(4000)	varchar(4000)	Operating system running on the machine.
arch	varchar(4000)	varchar(4000)	Architecture of the machine.
state	varchar(4000)	varchar(4000)	Condor state of the machine.
activity	varchar(4000)	varchar(4000)	Condor job activity on the machine.
keyboardidle	integer	integer	Number of seconds since activity has been detected on any keyboard or mouse associated with the machine.
consoleidle	integer	integer	Number of seconds since activity has been detected on the console keyboard or mouse.
loadavg	real	real	Current load average of the machine.
condorloadavg	real	real	Portion of load average generated by Condor
totalloadavg	real	real	
virtualmemory	integer	integer	Amount of currently available virtual memory in kilobytes.
memory	integer	integer	Amount of RAM in megabytes.
totalvirtualmemory	integer	integer	
cpubusytime	integer	integer	Time in seconds since cpuisbusy became true.
cpuisbusy	varchar(5)	varchar(5)	(boolean) True when the CPU is busy.
currentrank	real	real	The machine owner's affinity for running the Condor job which it is currently hosting.
clockmin	integer	integer	Number of minutes passed since midnight.
clockday	integer	integer	The day of the week.
lastreportedtime	ts(3) w tz	ts(3) w tz	Time when the Condor central manager last received a status update from this machine.
enteredcurrentactivity	ts(3) w tz	ts(3) w tz	Time when the machine entered the current activity.
enteredcurrentstate	ts(3) w tz	ts(3) w tz	Time when the machine entered the current state.
updatesequencenumber	integer	integer	Each update includes a sequence number.
Table Continues on Next Page			

Attributes of machines_horizontal_history Table – Part 2 of 2			
Name	**O. Type**	**P. Type**	**Description**
updatestotal	integer	integer	The number of updates received from the daemon.
updatessequenced	integer	integer	The number of updates that were in order.
updateslost	integer	integer	The number of updates that were lost.
globaljobid	varchar(4000)	varchar(4000)	Unique global identifier for the job.
end_time	ts(3) w tz	ts(3) w tz	The end of when the ClassAd is valid.

Attributes of machines_vertical Table			
Name	**O. Type**	**P. Type**	**Description**
machine_id	varchar(4000)	varchar(4000)	Unique identifier of the machine.
attr	varchar(2000)	varchar(2000)	Attribute name.
val	clob	text	Attribute value.
start_time	ts(3) w tz	ts(3) w tz	Time when this attribute–value pair became valid.
PRIMARY KEY: (machine_id, attr)			
NOT NULL: machine_id and attr cannot be null			

Attributes of machines_vertical_history Table			
Name	**O. Type**	**P. Type**	**Description**
machine_id	varchar(4000)	varchar(4000)	Unique identifier of the machine.
attr	varchar(4000)	varchar(4000)	Attribute name.
val	clob	text	Attribute value.
start_time	ts(3) w tz	ts(3) w tz	Time when this attribute–value pair became valid.
end_time	ts(3) w tz	ts(3) w tz	Time when this attribute–value pair became invalid.

Matchmaking Tables

Attributes of matches Table			
Name	**O. Type**	**P. Type**	**Description**
match_time	ts(3) w tz	ts(3) w tz	Time the match was made.
username	varchar(4000)	varchar(4000)	User who submitted the job.
scheddname	varchar(4000)	varchar(4000)	Name of the schedd that the job is submitted to.
cluster_id	integer	integer	Cluster identifier for the job.
proc_id	integer	integer	Process identifier for the job.
globaljobid	varchar(4000)	varchar(4000)	Unique global identifier for the job.
machine_id	varchar(4000)	varchar(4000)	Identifier of the machine the job matched with.
remote_user	varchar(4000)	varchar(4000)	User that was preempted.
remote_priority	real	real	The preempted user's priority.

Attributes of rejects Table			
Name	**O. Type**	**P. Type**	**Description**
reject_time	ts(3) w tz	ts(3) w tz	Time when the job was rejected.
username	varchar(4000)	varchar(4000)	User who submitted the job.
scheddname	varchar(4000)	varchar(4000)	Name of the schedd that submitted the job.
cluster_id	integer	integer	Cluster identifier for the job.
proc_id	integer	integer	Process identifier for the job.
globaljobid	varchar(4000)	varchar(4000)	Unique global identifier for the job.

Runtime Tables

Attributes of events Table			
Name	**O. Type**	**P. Type**	**Description**
scheddname	varchar(4000)	varchar(4000)	Name of the schedd that submitted the job.
cluster_id	integer	integer	Cluster identifier for the job.
proc_id	integer	integer	Process identifier for the job.
globaljobid	varchar(4000)	varchar(4000)	Global identifier of the job that generated the event.
run_id	numeric(12,0)	numeric(12,0)	Identifier of the run that the event is associated with.
eventtype	integer	integer	Numeric type code of the event.
eventtime	ts(3) w tz	ts(3) w tz	Time the event occurred.
description	varchar(4000)	varchar(4000)	Description of the event.

Attributes of generic_messages Table			
Name	**O. Type**	**P. Type**	**Description**
eventtype	varchar(4000)	varchar(4000)	The type of event.
eventkey	varchar(4000)	varchar(4000)	The key of the event.
eventtime	ts(3) w tz	ts(3) w tz	The time of the event.
eventloc	varchar(4000)	varchar(4000)	The location of the event.
attname	varchar(4000)	varchar(4000)	The attribute name.
attval	clob	text	The attribute value.
attrtype	varchar(4000)	varchar(4000)	The attribute type.

Attributes of runs Table			
Name	**O. Type**	**P. Type**	**Description**
run_id	numeric(12)	numeric(12)	Unique identifier of the run.
machine_id	varchar(4000)	varchar(4000)	Identifier of the machine where the job ran.
scheddname	varchar(4000)	varchar(4000)	Name of the schedd that submitted the job.
cluster_id	integer	integer	Cluster identifier for the job.
proc_id	integer	integer	Process identifier for the job.
spid	integer	integer	Subprocess identifier for the job.
globaljobid	varchar(4000)	varchar(4000)	Identifier of the job that was run.
startts	ts(3) w tz	ts(3) w tz	Time when the job started.
endts	ts(3) w tz	ts(3) w tz	Time when the job ended.
endtype	smallint	smallint	The type of ending event.
endmessage	varchar(4000)	varchar(4000)	The ending message.
wascheckpointed	varchar(7)	varchar(7)	Whether the run was checkpointed.
imagesize	numeric(38)	numeric(38)	The image size of the executable.
runlocalusageuser	integer	integer	The time the job spent in usermode on execute machines (only standard universe).
runlocalusagesystem	integer	integer	The time the job was in system calls.
runremoteusageuser	integer	integer	The time the shadow spent working for the job.
runremoteusagesystem	integer	integer	The time the shadow spent in system calls for the job.
runbytessent	numeric(38)	numeric(38)	Number of bytes sent to the run.
runbytesreceived	numeric(38)	numeric(38)	Number of bytes received from the run.
PRIMARY KEY: run_id			
NOT NULL: run_id cannot be null			

System Tables

Attributes of dummy_single_row_table Table			
Name	**O. Type**	**P. Type**	**Description**
a	varchar(1)	varchar(1)	A dummy column.

Attributes of history_jobs_to_purge Table			
scheddname	varchar(4000)	varchar(4000)	Name of the schedd that submitted the job.
cluster_id	integer	integer	Cluster identifier for the job.
proc_id	integer	integer	Process identifier for the job.
globaljobid	varchar(4000)	varchar(4000)	Unique global identifier for the job.

Attributes of jobqueuepollinginfo Table			
Name	**O. Type**	**P. Type**	**Description**
scheddname	varchar(4000)	varchar(4000)	Name of the schedd that submitted the job.
last_file_mtime	integer	integer	The last modification time of the file.
last_file_size	numeric(38)	numeric(38)	The last size of the file in bytes.
last_next_cmd_offset	integer	integer	The last offset for the next command.
last_cmd_offset	integer	integer	The last offset of the current command.
last_cmd_type	smallint	smallint	The last type of command.
last_cmd_key	varchar(4000)	varchar(4000)	The last key of the command.
last_cmd_mytype	varchar(4000)	varchar(4000)	The last my ClassAd type of the command.
last_cmd_targettype	varchar(4000)	varchar(4000)	The last target ClassAd type.
last_cmd_name	varchar(4000)	varchar(4000)	The attribute name of the command.
last_cmd_value	varchar(4000)	varchar(4000)	The attribute value of the command.

3.13 Setting Up for Special Environments

The following sections describe how to set up Condor for use in special environments or configurations.

3.13.1 Using Condor with AFS

Configuration variables that allow machines to interact with and use a shared file system are given at section 3.3.7.

Limitations with AFS occur because Condor does not currently have a way to authenticate itself to AFS. This is true of the Condor daemons that would like to authenticate as the AFS user `condor`,

and of the *condor_shadow* which would like to authenticate as the user who submitted the job it is serving. Since neither of these things can happen yet, there are special things to do when interacting with AFS. Some of this must be done by the administrator(s) installing Condor. Other things must be done by Condor users who submit jobs.

AFS and Condor for Administrators

The largest result from the lack of authentication with AFS is that the directory defined by the configuration variable `LOCAL_DIR` and its subdirectories `log` and `spool` on each machine must be either writable to unauthenticated users, or must not be on AFS. Making these directories writable a *very* bad security hole, so it is *not* a viable solution. Placing `LOCAL_DIR` onto NFS is acceptable. To avoid AFS, place the directory defined for `LOCAL_DIR` on a local partition on each machine in the pool. This implies running *condor_configure* to install the release directory and configure the pool, setting the `LOCAL_DIR` variable to a local partition. When that is complete, log into each machine in the pool, and run *condor_init* to set up the local Condor directory.

The directory defined by `RELEASE_DIR`, which holds all the Condor binaries, libraries, and scripts, can be on AFS. None of the Condor daemons need to write to these files. They only need to read them. So, the directory defined by `RELEASE_DIR` only needs to be world readable in order to let Condor function. This makes it easier to upgrade the binaries to a newer version at a later date, and means that users can find the Condor tools in a consistent location on all the machines in the pool. Also, the Condor configuration files may be placed in a centralized location. This is what we do for the UW-Madison's CS department Condor pool, and it works quite well.

Finally, consider setting up some targeted AFS groups to help users deal with Condor and AFS better. This is discussed in the following manual subsection. In short, create an AFS group that contains all users, authenticated or not, but which is restricted to a given host or subnet. These should be made as host-based ACLs with AFS, but here at UW-Madison, we have had some trouble getting that working. Instead, we have a special group for all machines in our department. The users here are required to make their output directories on AFS writable to any process running on any of our machines, instead of any process on any machine with AFS on the Internet.

AFS and Condor for Users

The *condor_shadow* daemon runs on the machine where jobs are submitted. It performs all file system access on behalf of the jobs. Because the *condor_shadow* daemon is not authenticated to AFS as the user who submitted the job, the *condor_shadow* daemon will not normally be able to write any output. Therefore the directories in which the job will be creating output files will need to be world writable; they need to be writable by non-authenticated AFS users. In addition, the program's `stdout`, `stderr`, log file, and any file the program explicitly opens will need to be in a directory that is world-writable.

An administrator may be able to set up special AFS groups that can make unauthenticated access to the program's files less scary. For example, there is supposed to be a way for AFS to grant access to any unauthenticated process on a given host. If set up, write access need only be granted to

unauthenticated processes on the submit machine, as opposed to any unauthenticated process on the Internet. Similarly, unauthenticated read access could be granted only to processes running on the submit machine.

A solution to this problem is to not use AFS for output files. If disk space on the submit machine is available in a partition not on AFS, submit the jobs from there. While the *condor_shadow* daemon is not authenticated to AFS, it does run with the effective UID of the user who submitted the jobs. So, on a local (or NFS) file system, the *condor_shadow* daemon will be able to access the files, and no special permissions need be granted to anyone other than the job submitter. If the Condor daemons are not invoked as root however, the *condor_shadow* daemon will not be able to run with the submitter's effective UID, leading to a similar problem as with files on AFS.

3.13.2 Using Condor with the Hadoop File System

The Hadoop project is an Apache project, headquartered at http://hadoop.apache.org, which implements an open-source, distributed file system across a large set of machines. The file system proper is called the Hadoop File System, or HDFS, and there are several Hadoop-provided tools which use the file system, most notably databases and tools which use the map-reduce distributed programming style. Condor provides a way to manage the daemons which implement an HDFS, but no direct support for the high-level tools which run atop this file system. There are two types of daemons, which together create an instance of a Hadoop File System. The first is called the Name node, which is like the central manager for a Hadoop cluster. There is only one active Name node per HDFS. If the Name node is not running, no files can be accessed. The HDFS does not support fail over of the Name node, but it does support a hot-spare for the Name node, called the Backup node. Condor can configure one node to be running as a Backup node. The second kind of daemon is the Data node, and there is one Data node per machine in the distributed file system. As these are both implemented in Java, Condor cannot directly manage these daemons. Rather, Condor provides a small DaemonCore daemon, called *condor_hdfs*, which reads the Condor configuration file, responds to Condor commands like *condor_on* and *condor_off*, and runs the Hadoop Java code. It translates entries in the Condor configuration file to an XML format native to Condor. These configuration items are listed with the *condor_hdfs* daemon in section 3.3.23. So, to configure HDFS in Condor, the Condor configuration file should specify one machine in the pool to be the HDFS Name node, and others to be the Data nodes.

Once an HDFS is deployed, Condor jobs can directly use it in a vanilla universe job, by transferring input files directly from the HDFS by specifying a URL within the job's submit description file command **transfer_input_files**. See section 3.13.3 for the administrative details to set up transfers specified by a URL. It requires that a plug-in is accessible and defined to handle `hdfs` protocol transfers.

3.13.3 Enabling the Transfer of Files Specified by a URL

A vanilla universe's job input files or a vm universe VM image file may be specified by a URL, causing the execute machine to retrieve the files. This differs from the normal file transfer mechanism,

in which transfers are from the machine where the job is submitted to the machine where the job is executed.

The transfers are accomplished a *plug-in*, an executable or shell script that handles the task of file transfer. More than one plug-in may be specified. The plug-ins must be installed and available on every execute machine that may run a job specifying an input file with a URL.

URL transfers are enabled by default in the configuration of execute machines. Disabling URL transfers is accomplished by setting

```
ENABLE_URL_TRANSFERS = FALSE
```

In addition, a comma separated list giving the absolute path and name of all available plug-ins is specified as in the example:

```
FILETRANSFER_PLUGINS = /opt/condor/plugins/wget-plugin, \
                       /opt/condor/plugins/hdfs-plugin, \
                       /opt/condor/plugins/custom-plugin
```

The *condor_starter* invokes all listed plug-ins to determine their capabilities. Each may handle one or more protocols (scheme names). The plug-in's response to invocation identifies which protocols it can handle. When a URL transfer is specified by a job, the *condor_starter* invokes the proper one to do the transfer. If more than one plugin is capable of handling a particular protocol, then the last one within the list given by `FILETRANSFER_PLUGINS` is used.

Condor assumes that all plug-ins will respond in specific ways. To determine the capabilities of the plug-ins as to which protocols they handle, the *condor_starter* daemon invokes each plug-in giving it the command line argument **-classad**. In response to invocation with this command line argument, the plug-in must respond with an output of three ClassAd attributes. The first two are fixed:

```
PluginVersion = "0.1"
PluginType = "FileTransfer"
```

The third ClassAd attribute is `SupportedMethods`. This attribute is a string containing a comma separated list of the protocols that the plug-in handles. So, for example:

```
SupportedMethods = "http,ftp,file"
```

would identify that the three protocols described by `http`, `ftp`, and `ftp` are supported. These strings will match the protocol specification as given within a URL in a **transfer_input_files** command in a submit description file for a job.

When a job specifies a URL transfer, the plug-in is invoked, without the command line argument **-classad**. It will instead be given two other command line arguments. The first will be the URL of the file to retrieve. The second will be the absolute path identifying where to place the transferred file. The plug-in is expected to do the transfer, exiting with status 0 if the transfer was successful, and a non-zero status if the transfer was *not* successful.

Note that this functionality is not limited to a predefined set of protocols. New ones can be invented. As an invented example, the `zkm` transfer type writes random bytes to a file. The plug-in that handles `zkm` transfers would respond to invocation with the **-classad** command line argument with:

```
PluginVersion = "0.1"
PluginType = "FileTransfer"
SupportedMethods = "zkm"
```

And, then when a job requested that this plug-in be invoked, for the invented example:

```
transfer_intput_files = zkm://128/r-data
```

the plug-in will be invoked with a first command line argument of `zkm://128/r-data` and a second command line argument giving the full path along with the file name `r-data` as the location for the plug-in to write 128 bytes of random data.

3.13.4 Configuring Condor for Multiple Platforms

A single, global configuration file may be used for all platforms in a Condor pool, with only platform-specific settings placed in separate files. This greatly simplifies administration of a heterogeneous pool by allowing changes of platform-independent, global settings in one place, instead of separately for each platform. This is made possible by treating the `LOCAL_CONFIG_FILE` configuration variable as a list of files, instead of a single file. Of course, this only helps when using a shared file system for the machines in the pool, so that multiple machines can actually share a single set of configuration files.

With multiple platforms, put all platform-independent settings (the vast majority) into the regular `condor_config` file, which would be shared by all platforms. This global file would be the one that is found with the `CONDOR_CONFIG` environment variable, the user `condor`'s home directory, or `/etc/condor/condor_config`.

Then set the `LOCAL_CONFIG_FILE` configuration variable from that global configuration file to specify both a platform-specific configuration file and optionally, a local, machine-specific configuration file (this parameter is described in section 3.3.3 on "Condor-wide Configuration File Entries").

The order of file specification in the `LOCAL_CONFIG_FILE` configuration variable is important, because settings in files at the beginning of the list are overridden if the same settings occur in files later within the list. So, if specifying the platform-specific file and then the machine-specific file, settings in the machine-specific file would override those in the platform-specific file (as is likely desired).

Utilizing a Platform-Specific Configuration File

The name of platform-specific configuration files may be specified by using the ARCH and OPSYS configuration variables, as are defined automatically by Condor. For example, for 32-bit Intel Windows 7 machines and 64-bit Intel Linux machines, the files ought to be named:

```
condor_config.INTEL.WINNT61
condor_config.X86_64.LINUX
```

Then, assuming these files are in the directory defined by the ETC configuration macro, and machine-specific configuration files are in the same directory, named by each machine's host name, the LOCAL_CONFIG_FILE configuration macro should be:

```
LOCAL_CONFIG_FILE = $(ETC)/condor_config.$(ARCH).$(OPSYS), \
                    $(ETC)/$(HOSTNAME).local
```

Alternatively, when using AFS, an "@sys link" may be used to specify the platform-specific configuration file, and let AFS resolve this link differently on different systems. For example, consider a soft link named condor_config.platform that points to condor_config.@sys. In this case, the files might be named:

```
condor_config.i386_linux2
condor_config.platform -> condor_config.@sys
```

and the LOCAL_CONFIG_FILE configuration variable would be set to:

```
LOCAL_CONFIG_FILE = $(ETC)/condor_config.platform, \
                    $(ETC)/$(HOSTNAME).local
```

Platform-Specific Configuration File Settings

The configuration variables that are truly platform-specific are:

RELEASE_DIR Full path to to the installed Condor binaries. While the configuration files may be shared among different platforms, the binaries certainly cannot. Therefore, maintain separate release directories for each platform in the pool. See section 3.3.3 on "Condor-wide Configuration File Entries" for details.

MAIL The full path to the mail program. See section 3.3.3 on "Condor-wide Configuration File Entries" for details.

CONSOLE_DEVICES Which devices in /dev should be treated as console devices. See section 3.3.10 on "condor_startd Configuration File Entries" for details.

DAEMON_LIST Which daemons the *condor_master* should start up. The reason this setting is platform-specific is to distinguish the *condor_kbdd*. It is needed on many Linux and Windows machines, and it is not needed on other platforms. See section 3.3.9 on for details.

Reasonable defaults for all of these configuration variables will be found in the default configuration files inside a given platform's binary distribution (except the RELEASE_DIR, since the location of the Condor binaries and libraries is installation specific). With multiple platforms, use one of the condor_config files from either running *condor_configure* or from the <release_dir>/etc/examples/condor_config.generic file, take these settings out, save them into a platform-specific file, and install the resulting platform-independent file as the global configuration file. Then, find the same settings from the configuration files for any other platforms to be set up, and put them in their own platform-specific files. Finally, set the LOCAL_CONFIG_FILE configuration variable to point to the appropriate platform-specific file, as described above.

Not even all of these configuration variables are necessarily going to be different. For example, if an installed mail program understands the **-s** option in /usr/local/bin/mail on all platforms, the MAIL macro may be set to that in the global configuration file, and not define it anywhere else. For a pool with only Linux or Windows machines, the DAEMON_LIST will be the same for each, so there is no reason not to put that in the global configuration file.

Other Uses for Platform-Specific Configuration Files

It is certainly possible that an installation may want other configuration variables to be platform-specific as well. Perhaps a different policy is desired for one of the platforms. Perhaps different people should get the e-mail about problems with the different platforms. There is nothing hard-coded about any of this. What is shared and what should not shared is entirely configurable.

Since the LOCAL_CONFIG_FILE macro can be an arbitrary list of files, an installation can even break up the global, platform-independent settings into separate files. In fact, the global configuration file might only contain a definition for LOCAL_CONFIG_FILE, and all other configuration variables would be placed in separate files.

Different people may be given different permissions to change different Condor settings. For example, if a user is to be able to change certain settings, but nothing else, those settings may be placed in a file which was early in the LOCAL_CONFIG_FILE list, to give that user write permission on that file, then include all the other files after that one. In this way, if the user was trying to change settings she/he should not, they would simply be overridden.

This mechanism is quite flexible and powerful. For very specific configuration needs, they can probably be met by using file permissions, the LOCAL_CONFIG_FILE configuration variable, and imagination.

3.13.5 Full Installation of condor_compile

In order to take advantage of two major Condor features: checkpointing and remote system calls, users of the Condor system need to relink their binaries. Programs that are not relinked for Condor can run in Condor's "vanilla" universe just fine, however, they cannot checkpoint and migrate, or run on machines without a shared filesystem.

To relink your programs with Condor, we provide a special tool, *condor_compile*. As installed by default, *condor_compile* works with the following commands: *gcc*, *g++*, *g77*, *cc*, *acc*, *c89*, *CC*, *f77*, *fort77*, *ld*. On Solaris and Digital Unix, *f90* is also supported. See the *condor_compile*(1) man page for details on using *condor_compile*.

However, you can make *condor_compile* work transparently with all commands on your system whatsoever, including *make*.

The basic idea here is to replace the system linker (*ld*) with the Condor linker. Then, when a program is to be linked, the condor linker figures out whether this binary will be for Condor, or for a normal binary. If it is to be a normal compile, the old *ld* is called. If this binary is to be linked for condor, the script performs the necessary operations in order to prepare a binary that can be used with condor. In order to differentiate between normal builds and condor builds, the user simply places *condor_compile* before their build command, which sets the appropriate environment variable that lets the condor linker script know it needs to do its magic.

In order to perform this full installation of *condor_compile*, the following steps need to be taken:

1. Rename the system linker from ld to ld.real.
2. Copy the condor linker to the location of the previous ld.
3. Set the owner of the linker to root.
4. Set the permissions on the new linker to 755.

The actual commands that you must execute depend upon the system that you are on. The location of the system linker (*ld*), is as follows:

Operating System	Location of ld (ld-path)
Linux	/usr/bin
Solaris 2.X	/usr/ccs/bin
OSF/1 (Digital Unix)	/usr/lib/cmplrs/cc

On these platforms, issue the following commands (as root), where *ld-path* is replaced by the path to your system's *ld*.

```
mv /[ld-path]/ld /[ld-path]/ld.real
cp /usr/local/condor/lib/ld /[ld-path]/ld
chown root /[ld-path]/ld
chmod 755 /[ld-path]/ld
```

If you remove Condor from your system latter on, linking will continue to work, since the condor linker will always default to compiling normal binaries and simply call the real ld. In the interest of simplicity, it is recommended that you reverse the above changes by moving your ld.real linker back to it's former position as ld, overwriting the condor linker.

NOTE: If you ever upgrade your operating system after performing a full installation of *condor_compile*, you will probably have to re-do all the steps outlined above. Generally speaking, new versions or patches of an operating system might replace the system ld binary, which would undo the full installation of *condor_compile*.

3.13.6 The *condor_kbdd*

The Condor keyboard daemon (*condor_kbdd*) monitors X events on machines where the operating system does not provide a way of monitoring the idle time of the keyboard or mouse. On UNIX platforms, it is needed to detect USB keyboard activity but otherwise is not needed. On Windows the *condor_kbdd* is the primary method of monitoring both keyboard and mouse idleness.

With the move of user sessions out of session 0 on Windows Vista, the *condor_startd* service is no longer able to listen to keyboard and mouse events as all services run in session 0. As such, any execute node will require *condor_kbdd* to accurately monitor and report system idle time. This is achieved by auto-starting the *condor_kbdd* whenever a user logs into the system. The daemon will run in an invisible window and should not be noticeable by the user except for a listing in the task manager. When the user logs out, the program is terminated by Windows. This change has been made even to pre-Vista Windows versions because it adds the capability of monitoring keyboard activity from multiple users.

To achieve the auto-start with user login, the Condor installer adds a *condor_kbdd* entry to the registry key at `HKLM\Software\Microsoft\Windows\CurrentVersion\Run`. On 64bit versions of Vista and higher, the entry is actually placed in `HKLM\Software\Wow6432Node\Microsoft\Windows\CurrentVersion\Run`.
In instances where the *condor_kbdd* is unable to connect to the *condor_startd* on Windows XP SP2 or higher, it is likely because an exception was not properly added to the Windows firewall.

On UNIX, great measures have been taken to make this daemon as robust as possible, but the X window system was not designed to facilitate such a need, and thus is less then optimal on machines where many users log in and out on the console frequently.

In order to work with X authority, the system by which X authorizes processes to connect to X servers, the *condor_kbdd* needs to run with super user privileges. Currently, the daemon assumes that X uses the `HOME` environment variable in order to locate a file named `.Xauthority`, which contains keys necessary to connect to an X server. The keyboard daemon attempts to set this environment variable to various users home directories in order to gain a connection to the X server and monitor events. This may fail to work on your system, if you are using a non-standard approach. If the keyboard daemon is not allowed to attach to the X server, the state of a machine may be incorrectly set to idle when a user is, in fact, using the machine.

In some environments, the *condor_kbdd* will not be able to connect to the X server because

the user currently logged into the system keeps their authentication token for using the X server in a place that no local user on the current machine can get to. This may be the case for AFS where the user's `.Xauthority` file is in an AFS home directory. There may also be cases where the *condor_kbdd* may not be run with super user privileges because of political reasons, but it is still desired to be able to monitor X activity. In these cases, change the XDM configuration in order to start up the *condor_kbdd* with the permissions of the currently logging in user. Although your situation may differ, if you are running X11R6.3, you will probably want to edit the files in `/usr/X11R6/lib/X11/xdm`. The `.xsession` file should have the keyboard daemon start up at the end, and the `.Xreset` file should have the keyboard daemon shut down. The **-l** option can be used to write the daemon's log file to a place where the user running the daemon has permission to write a file. We recommend something akin to `$HOME/.kbdd.log`, since this is a place where every user can write, and it will not get in the way. The **-pidfile** and **-k** options allow for easy shut down of the daemon by storing the process id in a file. It will be necessary to add lines to the XDM configuration that look something like:

```
condor_kbdd -l $HOME/.kbdd.log -pidfile $HOME/.kbdd.pid
```

This will start the *condor_kbdd* as the user who is currently logging in and write the log to a file in the directory `$HOME/.kbdd.log/`. Also, this will save the process id of the daemon to `~/.kbdd.pid`, so that when the user logs out, XDM can do:

```
condor_kbdd -k $HOME/.kbdd.pid
```

This will shut down the process recorded in `~/.kbdd.pid` and exit.

To see how well the keyboard daemon is working, review the log for the daemon and look for successful connections to the X server. If there are none, the *condor_kbdd* is unable to connect to the machine's X server.

3.13.7 Configuring The CondorView Server

The CondorView server is an alternate use of the *condor_collector* that logs information on disk, providing a persistent, historical database of pool state. This includes machine state, as well as the state of jobs submitted by users.

An existing *condor_collector* may act as the CondorView collector through configuration. This is the simplest situation, because the only change needed is to turn on the logging of historical information. The alternative of configuring a new *condor_collector* to act as the CondorView collector is slightly more complicated, while it offers the advantage that the same CondorView collector may be used for several pools as desired, to aggregate information into one place.

The following sections describe how to configure a machine to run a CondorView server and to configure a pool to send updates to it.

Configuring a Machine to be a CondorView Server

To configure the CondorView collector, a few configuration variables are added or modified for the *condor_collector* chosen to act as the CondorView collector. These configuration variables are described in section 3.3.16 on page 224. Here are brief explanations of the entries that must be customized:

POOL_HISTORY_DIR The directory where historical data will be stored. This directory must be writable by whatever user the CondorView collector is running as (usually the user `condor`). There is a configurable limit to the maximum space required for all the files created by the CondorView server called (`POOL_HISTORY_MAX_STORAGE`).

NOTE: This directory should be separate and different from the `spool` or `log` directories already set up for Condor. There are a few problems putting these files into either of those directories.

KEEP_POOL_HISTORY A boolean value that determines if the CondorView collector should store the historical information. It is `False` by default, and must be specified as `True` in the local configuration file to enable data collection.

Once these settings are in place in the configuration file for the CondorView server host, create the directory specified in `POOL_HISTORY_DIR` and make it writable by the user the CondorView collector is running as. This is the same user that owns the `CollectorLog` file in the `log` directory. The user is usually `condor`.

If using the existing *condor_collector* as the CondorView collector, no further configuration is needed. To run a different *condor_collector* to act as the CondorView collector, configure Condor to automatically start it.

If using a separate host for the CondorView collector, to start it, add the value `COLLECTOR` to `DAEMON_LIST`, and restart Condor on that host. To run the CondorView collector on the same host as another *condor_collector*, ensure that the two *condor_collector* daemons use different network ports. Here is an example configuration in which the main *condor_collector* and the CondorView collector are started up by the same *condor_master* daemon on the same machine. In this example, the CondorView collector uses port 12345.

```
VIEW_SERVER = $(COLLECTOR)
VIEW_SERVER_ARGS = -f -p 12345
VIEW_SERVER_ENVIRONMENT = "_CONDOR_COLLECTOR_LOG=$(LOG)/ViewServerLog"
DAEMON_LIST = MASTER, NEGOTIATOR, COLLECTOR, VIEW_SERVER
```

For this change to take effect, restart the *condor_master* on this host. This may be accomplished with the *condor_restart* command, if the command is run with administrator access to the pool.

Configuring a Pool to Report to the CondorView Server

For the CondorView server to function, configure the existing collector to forward ClassAd updates to it. This configuration is only necessary if the CondorView collector is a different collector from the existing *condor_collector* for the pool. All the Condor daemons in the pool send their ClassAd updates to the regular *condor_collector*, which in turn will forward them on to the CondorView server.

Define the following configuration variable:

```
CONDOR_VIEW_HOST = full.hostname[:portnumber]
```

where `full.hostname` is the full host name of the machine running the CondorView collector. The full host name is optionally followed by a colon and port number. This is only necessary if the CondorView collector is configured to use a port number other than the default.

Place this setting in the configuration file used by the existing *condor_collector*. It is acceptable to place it in the global configuration file. The CondorView collector will ignore this setting (as it should) as it notices that it is being asked to forward ClassAds to itself.

Once the CondorView server is running with this change, send a *condor_reconfig* command to the main *condor_collector* for the change to take effect, so it will begin forwarding updates. A query to the CondorView collector will verify that it is working. A query example:

```
condor_status -pool condor.view.host[:portnumber]
```

3.13.8 Running Condor Jobs within a Virtual Machine

Condor jobs are formed from executables that are compiled to execute on specific platforms. This in turn restricts the machines within a Condor pool where a job may be executed. A Condor job may now be executed on a virtual machine system running VMware, Xen, or KVM. This allows Windows executables to run on a Linux machine, and Linux executables to run on a Windows machine.

In older versions of Condor, other parts of the system were also referred to as *virtual machines*, but in all cases, those are now known as *slots*. A virtual machine here describes the environment in which the outside operating system (called the host) emulates an inner operating system (called the inner virtual machine), such that an executable appears to run directly on the inner virtual machine. In other parts of Condor, a *slot* (formerly known as *virtual machine*) refers to the multiple CPUs of an SMP machine. Also, be careful not to confuse the virtual machines discussed here with the Java Virtual Machine (JVM) referenced in other parts of this manual.

Condor has the flexibility to run a job on either the host or the inner virtual machine, hence two platforms appear to exist on a single machine. Since two platforms are an illusion, Condor understands the illusion, allowing a Condor job to be execute on only one at a time.

Installation and Configuration

Condor must be separately installed, separately configured, and separately running on both the host and the inner virtual machine.

The configuration for the host specifies VMP_VM_LIST . This specifies host names or IP addresses of all inner virtual machines running on this host. An example configuration on the host machine:

```
VMP_VM_LIST = vmware1.domain.com, vmware2.domain.com
```

The configuration for each separate inner virtual machine specifies VMP_HOST_MACHINE . This specifies the host for the inner virtual machine. An example configuration on an inner virtual machine:

```
VMP_HOST_MACHINE = host.domain.com
```

Given this configuration, as well as communication between Condor daemons running on the host and on the inner virtual machine, the policy for when jobs may execute is set by Condor. While the host is executing a Condor job, the START policy on the inner virtual machine is overridden with False, so no Condor jobs will be started on the inner virtual machine. Conversely, while the inner virtual machine is executing a Condor job, the START policy on the host is overridden with False, so no Condor jobs will be started on the host.

The inner virtual machine is further provided with a new syntax for referring to the machine ClassAd attributes of its host. Any machine ClassAd attribute with a prefix of the string HOST_ explicitly refers to the host's ClassAd attributes. The START policy on the inner virtual machine ought to use this syntax to avoid starting jobs when its host is too busy processing other items. An example configuration for START on an inner virtual machine:

```
START = ( (KeyboardIdle > 150 ) && ( HOST_KeyboardIdle > 150 ) \
        && ( LoadAvg <= 0.3 ) && ( HOST_TotalLoadAvg <= 0.3 ) )
```

3.13.9 Configuring The Startd for SMP Machines

This section describes how to configure the *condor_startd* for SMP (Symmetric Multi-Processor) machines. Machines with more than one CPU may be configured to run more than one job at a time. As always, owners of the resources have great flexibility in defining the policy under which multiple jobs may run, suspend, vacate, etc.

How Shared Resources are Represented to Condor

The way SMP machines are represented to the Condor system is that the shared resources are broken up into individual *slots*. Each slot can be matched and claimed by users. Each slot is represented

by an individual ClassAd (see the ClassAd reference, section 4.1, for details). In this way, each SMP machine will appear to the Condor system as a collection of separate slots. As an example, an SMP machine named vulture.cs.wisc.edu would appear to Condor as the multiple machines, named slot1@vulture.cs.wisc.edu, slot2@vulture.cs.wisc.edu, slot3@vulture.cs.wisc.edu, and so on.

The way that the *condor_startd* breaks up the shared system resources into the different slots is configurable. All shared system resources (like RAM, disk space, swap space, etc.) can either be divided evenly among all the slots, with each CPU getting its own slot, or you can define your own *slot types*, so that resources can be unevenly partitioned. Regardless of the partitioning scheme used, it is important to remember the goal is to create a representative slot ClassAd, to be used for matchmaking with jobs. Condor does not directly enforce slot shared resource allocations, and jobs are free to oversubscribe to shared resources.

Consider an example where two slots are each defined with 50%of available RAM. The resultant ClassAd for each slot will advertise one half the available RAM. Users may submit jobs with RAM requirements that match these slots. However, jobs run on either slot are free to consume more than 50%of available RAM. Condor will not directly enforce a RAM utilization limit on either slot. If a shared resource enforcement capability is needed, it is possible to write a Startd policy that will evict a job that oversubscribes to shared resources, see section 3.13.9.

The following section gives details on how to configure Condor to divide the resources on an SMP machine into separate slots.

Dividing System Resources in SMP Machines

This section describes the settings that allow you to define your own slot types and to control how many slots of each type are reported to Condor.

There are two main ways to go about partitioning an SMP machine:

Define your own slot types. By defining your own types, you can specify what fraction of shared system resources (CPU, RAM, swap space and disk space) go to each slot. Once you define your own types, you can control how many of each type are reported at any given time.

Evenly divide all resources. If you do not define your own types, the *condor_startd* will automatically partition your machine into slots for you. It will do so by placing a single CPU in each slot, and evenly dividing all shared resources among the slots. With this default partitioning, you only specify how many slots are reported at a time. By default, all slots are reported to Condor.

The number of each type being reported can be changed at run-time, by issuing a reconfiguration command to the *condor_startd* daemon (sending a SIGHUP or using *condor_reconfig*). However, the definitions for the types themselves cannot be changed with reconfiguration. If you change any slot type definitions, you must use *condor_restart*

```
condor_restart -startd
```

for that change to take effect.

Defining Slot Types

To define your own slot types, add configuration file parameters that list how much of each system resource you want in the given slot type. Do this by defining configuration variables of the form SLOT_TYPE_<N> . The <N> represents an integer (for example, SLOT_TYPE_1), which specifies the slot type defined. Note that there may be multiple slots of each type. The number created is configured with NUM_SLOTS_TYPE_<N> as described later in this section.

A type describes what share of the total system resources a given slot has available to it.

The type can be defined by:

- A simple fraction, such as 1/4
- A simple percentage, such as 25%
- A comma-separated list of attributes, with a percentage, fraction, numerical value, or auto for each one.
- A comma-separated list including a blanket value that serves as a default for any resources not explicitly specified in the list.

A simple fraction or percentage causes an allocation of the total system resources. This includes the number of CPUs. A comma-separated list allows a fine-tuning of the amounts for specific attributes.

The attributes that specify the number of CPUs and the total amount of RAM in the SMP machine do not change. For these attributes, specify either absolute values or percentages of the total available amount (or auto). For example, in a machine with 128 Mbytes of RAM, all the following definitions result in the same allocation amount.

```
mem=64
mem=1/2
mem=50%
mem=auto
```

Other attributes are dynamic, such as disk space and swap space. For these, specify a percentage or fraction of the total value that is allocated to each slot, instead of specifying absolute values. As the total values of these resources change on your machine, each slot will take its fraction of the total and report that as its available amount.

The disk space allocated to each slot is taken from the disk partition containing the slots execute directory (configured with EXECUTE or SLOT<N>_EXECUTE). If every slot is in a different partition, then each one may be defined with up to 100%for its disk share. If some slots are in the same partition, then their total is not allowed to exceed 100%.

The four attribute names are case insensitive when defining slot types. The first letter of the attribute name distinguishes between the attributes. The four attributes, with several examples of acceptable names for each are

- Cpus, C, c, cpu
- ram, RAM, MEMORY, memory, Mem, R, r, M, m
- disk, Disk, D, d
- swap, SWAP, S, s, VirtualMemory, V, v

As an example, consider a host of 4 CPUs and 256 megs of RAM. Here are valid example slot type definitions. Types 1-3 are all equivalent to each other, as are types 4-6. Note that in a real configuration, you would not use all of these slot types together because they add up to more than 100%of the various system resources. Also note that in a real configuration, you would need to also define `NUM_SLOTS_TYPE_<N>` for each slot type.

```
SLOT_TYPE_1 = cpus=2, ram=128, swap=25%, disk=1/2

SLOT_TYPE_2 = cpus=1/2, memory=128, virt=25%, disk=50%

SLOT_TYPE_3 = c=1/2, m=50%, v=1/4, disk=1/2

SLOT_TYPE_4 = c=25%, m=64, v=1/4, d=25%

SLOT_TYPE_5 = 25%

SLOT_TYPE_6 = 1/4
```

The default value for each resource share is `auto`. The share may also be explicitly set to `auto`. All slots with the value `auto` for a given type of resource will evenly divide whatever remains after subtracting out whatever was explicitly allocated in other slot definitions. For example, if one slot is defined to use 10%of the memory and the rest define it as `auto` (or leave it undefined), then the rest of the slots will evenly divide 90%of the memory between themselves.

In both of the following examples, the disk share is set to `auto`, cpus is 1, and everything else is 50%:

```
SLOT_TYPE_1 = cpus=1, ram=1/2, swap=50%

SLOT_TYPE_1 = cpus=1, disk=auto, 50%
```

The number of slots of each type is set with the configuration variable `NUM_SLOTS_TYPE_<N>` , where N is the type as given in the `SLOT_TYPE_<N>`variable.

Note that it is possible to set the configuration variables such that they specify an impossible configuration. If this occurs, the *condor_startd* daemon fails after writing a message to its log attempting to indicate the configuration requirements that it could not implement.

Evenly Divided Resources

If you are not defining your own slot types, then all resources are divided equally among the slots. The number of slots within the SMP machine is the only attribute that needs to be defined. Its definition is accomplished by setting the configuration variable `NUM_SLOTS` to the integer number of slots desired. If variable `NUM_SLOTS` is not defined, it defaults to the number of CPUs within the SMP machine. You cannot use `NUM_SLOTS` to make Condor advertise more slots than there are CPUs on the machine. To do that, use `NUM_CPUS` .

Configuring Startd Policy for SMP Machines

Section 3.5 details the Startd Policy Configuration. This section continues the discussion with respect to SMP machines.

Each slot within an SMP machine is treated as an independent machine, each with its own view of its machine state. There is a single set of policy expressions for the SMP machine as a whole. This policy may consider the slot state(s) in its expressions. This makes some policies easy to set, but it makes other policies difficult or impossible to set.

An easy policy to set configures how many of the slots notice console or tty activity on the SMP as a whole. Slots that are not configured to notice any activity will report ConsoleIdle and KeyboardIdle times from when the *condor_startd* daemon was started, (plus a configurable number of seconds). With this, you can set up a multiple CPU machine with the default policy settings plus add that the keyboard and console noticed by only one slot. Assuming a reasonable load average (see section 3.13.9 below on "Load Average for SMP Machines"), only the one slot will suspend or vacate its job when the owner starts typing at their machine again. The rest of the slots could be matched with jobs and leave them running, even while the user was interactively using the machine. If the default policy is used, all slots notice tty and console activity and currently running jobs would suspend or preempt.

This example policy is controlled with the following configuration variables.

- `SLOTS_CONNECTED_TO_CONSOLE`
- `SLOTS_CONNECTED_TO_KEYBOARD`
- `DISCONNECTED_KEYBOARD_IDLE_BOOST`

These configuration variables are fully described in section 3.3.10 on page 195 which lists all the configuration file settings for the *condor_startd*.

The configuration of slots allows each slot to advertise its own machine ClassAd. Yet, there is only one set of policy expressions for the SMP machine as a whole. This makes the implementation of certain types of policies impossible. While evaluating the state of one slot (within the SMP machine), the state of other slots (again within the SMP machine) are not available. Decisions for one slot cannot be based on what other machines within the SMP are doing.

Specifically, the evaluation of a slot policy expression works in the following way.

1. The configuration file specifies policy expressions that are shared among all of the slots on the SMP machine.
2. Each slot reads the configuration file and sets up its own machine ClassAd.
3. Each slot is now separate from the others. It has a different state, a different machine ClassAd, and if there is a job running, a separate job ad. Each slot periodically evaluates the policy expressions, changing its own state as necessary. This occurs independently of the other slots on the machine. So, if the *condor_startd* daemon is evaluating a policy expression on a specific slot, and the policy expression refers to `ProcID`, `Owner`, or any attribute from a job ad, it *always* refers to the ClassAd of the job running on the specific slot.

To set a different policy for the slots within an SMP machine, a (`SUSPEND`) policy will be of the form

```
SUSPEND = ( (SlotID == 1) && (PolicyForSlot1) ) || \
            ( (SlotID == 2) && (PolicyForSlot2) )
```

where `(PolicyForSlot1)` and `(PolicyForSlot2)` are the desired expressions for each slot.

Load Average for SMP Machines

Most operating systems define the load average for an SMP machine as the total load on all CPUs. For example, if you have a 4-CPU machine with 3 CPU-bound processes running at the same time, the load would be 3.0 In Condor, we maintain this view of the total load average and publish it in all resource ClassAds as `TotalLoadAvg`.

Condor also provides a per-CPU load average for SMP machines. This nicely represents the model that each node on an SMP is a slot, separate from the other nodes. All of the default, single-CPU policy expressions can be used directly on SMP machines, without modification, since the `LoadAvg` and `CondorLoadAvg` attributes are the per-slot versions, not the total, SMP-wide versions.

The per-CPU load average on SMP machines is a Condor invention. No system call exists to ask the operating system for this value. Condor already computes the load average generated by Condor on each slot. It does this by close monitoring of all processes spawned by any of the Condor daemons, even ones that are orphaned and then inherited by *init*. This Condor load average per

slot is reported as the attribute `CondorLoadAvg` in all resource ClassAds, and the total Condor load average for the entire machine is reported as `TotalCondorLoadAvg`. The total, system-wide load average for the entire machine is reported as `TotalLoadAvg`. Basically, Condor walks through all the slots and assigns out portions of the total load average to each one. First, Condor assigns the known Condor load average to each node that is generating load. If there's any load average left in the total system load, it is considered an owner load. Any slots Condor believes are in the Owner state (like ones that have keyboard activity), are the first to get assigned this owner load. Condor hands out owner load in increments of at most 1.0, so generally speaking, no slot has a load average above 1.0. If Condor runs out of total load average before it runs out of virtual machines, all the remaining machines believe that they have no load average at all. If, instead, Condor runs out of slots and it still has owner load remaining, Condor starts assigning that load to Condor nodes as well, giving individual nodes with a load average higher than 1.0.

Debug logging in the SMP Startd

This section describes how the *condor_startd* daemon handles its debugging messages for SMP machines. In general, a given log message will either be something that is machine-wide (like reporting the total system load average), or it will be specific to a given slot. Any log entrees specific to a slot have an extra header printed out in the entry: `slot#:`. So, for example, here's the output about system resources that are being gathered (with `D_FULLDEBUG` and `D_LOAD` turned on) on a 2-CPU machine with no Condor activity, and the keyboard connected to both slots:

```
11/25 18:15 Swap space: 131064
11/25 18:15 number of Kbytes available for (/home/condor/execute): 1345063
11/25 18:15 Looking up RESERVED_DISK parameter
11/25 18:15 Reserving 5120 Kbytes for file system
11/25 18:15 Disk space: 1339943
11/25 18:15 Load avg: 0.340000 0.800000 1.170000
11/25 18:15 Idle Time: user= 0 , console= 4 seconds
11/25 18:15 SystemLoad: 0.340   TotalCondorLoad: 0.000  TotalOwnerLoad: 0.340
11/25 18:15 slot1: Idle time: Keyboard: 0        Console: 4
11/25 18:15 slot1: SystemLoad: 0.340  CondorLoad: 0.000  OwnerLoad: 0.340
11/25 18:15 slot2: Idle time: Keyboard: 0        Console: 4
11/25 18:15 slot2: SystemLoad: 0.000  CondorLoad: 0.000  OwnerLoad: 0.000
11/25 18:15 slot1: State: Owner           Activity: Idle
11/25 18:15 slot2: State: Owner           Activity: Idle
```

If, on the other hand, this machine only had one slot connected to the keyboard and console, and the other slot was running a job, it might look something like this:

```
11/25 18:19 Load avg: 1.250000 0.910000 1.090000
11/25 18:19 Idle Time: user= 0 , console= 0 seconds
11/25 18:19 SystemLoad: 1.250   TotalCondorLoad: 0.996  TotalOwnerLoad: 0.254
11/25 18:19 slot1: Idle time: Keyboard: 0        Console: 0
```

```
11/25 18:19 slot1: SystemLoad: 0.254  CondorLoad: 0.000  OwnerLoad: 0.254
11/25 18:19 slot2: Idle time: Keyboard: 1496     Console: 1496
11/25 18:19 slot2: SystemLoad: 0.996  CondorLoad: 0.996  OwnerLoad: 0.000
11/25 18:19 slot1: State: Owner           Activity: Idle
11/25 18:19 slot2: State: Claimed         Activity: Busy
```

As you can see, shared system resources are printed without the header (like total swap space), and slot-specific messages (like the load average or state of each slot) get the special header appended.

Configuring STARTD_ATTRS on a per-slot basis

The STARTD_ATTRS (and legacy STARTD_EXPRS) settings can be configured on a per-slot basis. The *condor_startd* daemon builds the list of items to advertise by combining the lists in this order:

1. STARTD_ATTRS
2. STARTD_EXPRS
3. SLOT<N>_STARTD_ATTRS
4. SLOT<N>_STARTD_EXPRS

For example, consider the following configuration:

```
STARTD_ATTRS = favorite_color, favorite_season
SLOT1_STARTD_ATTRS = favorite_movie
SLOT2_STARTD_ATTRS = favorite_song
```

This will result in the *condor_startd* ClassAd for slot1 defining values for favorite_color, favorite_season, and favorite_movie. slot2 will have values for favorite_color, favorite_season, and favorite_song.

Attributes themselves in the STARTD_ATTRS list can also be defined on a per-slot basis. Here is another example:

```
favorite_color = "blue"
favorite_season = "spring"
STARTD_ATTRS = favorite_color, favorite_season
SLOT2_favorite_color = "green"
SLOT3_favorite_season = "summer"
```

For this example, the *condor_startd* ClassAds are

slot1:

```
favorite_color = "blue"
favorite_season = "spring"
```

slot2:

```
favorite_color = "green"
favorite_season = "spring"
```

slot3:

```
favorite_color = "blue"
favorite_season = "summer"
```

Dynamic *condor_startd* Provisioning: Dynamic Slots

Dynamic provisioning, also referred to as a partitionable *condor_startd* or as dynamic slots, allows users to mark slots as partitionable. This means that more than one job can occupy a single slot at any one time. Typically, slots have a fixed set of resources, including the CPUs, memory and disk space. By partitioning the slot, these resources become more flexible and able to be better utilized.

Dynamic provisioning provides powerful configuration possibilities, and so should be used with care. Specifically, while preemption occurs for each individual dynamic slot, it cannot occur directly for the partitionable slot, or for groups of dynamic slots. For example, for a large number of jobs requiring 1GB of memory, a pool might be split up into 1GB dynamic slots. In this instance a job requiring 2GB of memory will be starved and unable to run.

Here is an example that demonstrates how more than one job can be matched to a single slot using dynamic provisioning. In this example, slot1 has the following resources:

cpu=10

memory=10240

disk=BIG

Assume that JobA is allocated to this slot. JobA includes the following requirements:

cpu=3

memory=1024

disk=10240

The portion of the slot that is utilized is referred to as Slot1.1, and after allocation, the slot advertises that it has the following resources still available:

cpu=7

memory=9216

disk=BIG-10240

As each new job is allocated to Slot1, it breaks into Slot1.1, Slot1.2, etc., until the entire set of available resources have been consumed by jobs.

To enable dynamic provisioning, set the SLOT_TYPE_<N>_PARTITIONABLE configuration variable to True. The string N within the configuration variable name is the slot number.

In a pool using dynamic provisioning, jobs can have extra, and desired, resources specified in the submit description file:

request_cpus

request_memory

request_disk (in kilobytes)

This example shows a portion of the job submit description file for use when submitting a job to a pool with dynamic provisioning.

```
universe = vanilla

request_cpus = 3
request_memory = 1024
request_disk = 10240

queue
```

For each type of slot, the original, partitionable slot and the new smaller, dynamic slots, an attribute is added to identify it. The original slot, as defined at page 915, will have an attribute stating

```
PartitionableSlot = True
```

and the dynamic slots will have an attribute, as defined at page 914,

```
DynamicSlot = True
```

These attributes may be used in a START expression for the purposes of creating detailed policies.

A partitionable slot will always appear as though it is not running a job. It will eventually show as having no available resources, which will prevent further matching to new jobs. Because it has been effectively broken up into smaller slots, these will show as running jobs directly. These dynamic slots can also be preempted in the same way as nonpartitioned slots.

3.13.10 Condor's Dedicated Scheduling

The dedicated scheduler is a part of the *condor_schedd* that handles the scheduling of parallel jobs that require more than one machine concurrently running per job. MPI applications are a common use for the dedicated scheduler, but parallel applications which do not require MPI can also be run with the dedicated scheduler. All jobs which use the parallel universe are routed to the dedicated scheduler within the *condor_schedd* they were submitted to. A default Condor installation does not configure a dedicated scheduler; the administrator must designate one or more *condor_schedd* daemons to perform as dedicated scheduler.

Selecting and Setting Up a Dedicated Scheduler

We recommend that you select a single machine within a Condor pool to act as the dedicated scheduler. This becomes the machine from upon which all users submit their parallel universe jobs. The perfect choice for the dedicated scheduler is the single, front-end machine for a dedicated cluster of compute nodes. For the pool without an obvious choice for a submit machine, choose a machine that all users can log into, as well as one that is likely to be up and running all the time. All of Condor's other resource requirements for a submit machine apply to this machine, such as having enough disk space in the spool directory to hold jobs. See section 3.2.2 on page 128 for details on these issues.

Configuration Examples for Dedicated Resources

Each machine may have its own policy for the execution of jobs. This policy is set by configuration. Each machine with aspects of its configuration that are dedicated identifies the dedicated scheduler. And, the ClassAd representing a job to be executed on one or more of these dedicated machines includes an identifying attribute. An example configuration file with the following various policy settings is `/etc/condor_config.local.dedicated.resource`.

Each dedicated machine defines the configuration variable `DedicatedScheduler` , which identifies the dedicated scheduler it is managed by. The local configuration file for any dedicated resource contains a modified form of

```
DedicatedScheduler = "DedicatedScheduler@full.host.name"
STARTD_ATTRS = $(STARTD_ATTRS), DedicatedScheduler
```

Substitute the host name of the dedicated scheduler machine for the string "`full.host.name`".

If running personal Condor, the name of the scheduler includes the user name it was started as, so the configuration appears as:

```
DedicatedScheduler = "DedicatedScheduler@username@full.host.name"
STARTD_ATTRS = $(STARTD_ATTRS), DedicatedScheduler
```

All dedicated resources must have policy expressions which allow for jobs to always run, but not be preempted. The resource must also be configured to prefer jobs from the dedicated scheduler over all other jobs. Therefore, configuration gives the dedicated scheduler of choice the highest rank. It is worth noting that Condor puts no other requirements on a resource for it to be considered dedicated.

Job ClassAds from the dedicated scheduler contain the attribute `Scheduler`. The attribute is defined by a string of the form

```
Scheduler = "DedicatedScheduler@full.host.name"
```

The host name of the dedicated scheduler substitutes for the string `full.host.name`.

Different resources in the pool may have different dedicated policies by varying the local configuration.

Policy Scenario: Machine Runs Only Jobs That Require Dedicated Resources One possible scenario for the use of a dedicated resource is to only run jobs that require the dedicated resource. To enact this policy, the configure with the following expressions:

```
START     = Scheduler =?= $(DedicatedScheduler)
SUSPEND   = False
CONTINUE  = True
PREEMPT   = False
KILL      = False
WANT_SUSPEND   = False
WANT_VACATE    = False
RANK      = Scheduler =?= $(DedicatedScheduler)
```

The `START` expression specifies that a job with the `Scheduler` attribute must match the string corresponding `DedicatedScheduler` attribute in the machine ClassAd. The `RANK` expression specifies that this same job (with the `Scheduler` attribute) has the highest rank. This prevents other jobs from preempting it based on user priorities. The rest of the expressions disable all of the *condor_startd* daemon's regular policies for evicting jobs when keyboard and CPU activity is discovered on the machine.

Policy Scenario: Run Both Jobs That Do and Do Not Require Dedicated Resources While the first example works nicely for jobs requiring dedicated resources, it can lead to poor utilization of the dedicated machines. A more sophisticated strategy allows the machines to run other jobs, when no jobs that require dedicated resources exist. The machine is configured to prefer jobs that require dedicated resources, but not prevent others from running.

To implement this, configure the machine as a dedicated resource (as above) modifying only the `START` expression:

```
START = True
```

Policy Scenario: Adding Desk-Top Resources To The Mix A third policy example allows all jobs. These desk-top machines use a preexisting START expression that takes the machine owner's usage into account for some jobs. The machine does not preempt jobs that must run on dedicated resources, while it will preempt other jobs based on a previously set policy. So, the default pool policy is used for starting and stopping jobs, while jobs that require a dedicated resource always start and are not preempted.

The START, SUSPEND, PREEMPT, and RANK policies are set in the global configuration. Locally, the configuration is modified to this hybrid policy by adding a second case.

```
SUSPEND    = Scheduler =!= $(DedicatedScheduler) && ($(SUSPEND))
PREEMPT    = Scheduler =!= $(DedicatedScheduler) && ($(PREEMPT))
RANK_FACTOR    = 1000000
RANK   = (Scheduler =?= $(DedicatedScheduler) * $(RANK_FACTOR)) \
               + $(RANK)
START  = (Scheduler =?= $(DedicatedScheduler)) || ($(START))
```

Define RANK_FACTOR to be a larger value than the maximum value possible for the existing rank expression. RANK is just a floating point value, so there is no harm in having a value that is very large.

Policy Scenario: Parallel Scheduling Groups In some parallel environments, machines are divided into groups, and jobs should not cross groups of machines – that is, all the nodes of a parallel job should be allocated to machines within the same group. The most common example is a pool of machines using infiniband switches. Each switch might connect 16 machines, and a pool might have 160 machines on 10 switches. If the infiniband switches are not routed to each other, each job must run on machines connected to the same switch.

The dedicated scheduler's parallel scheduling groups features supports jobs that must not cross group boundaries. Define a group by having each machine within a group set the configuration variable ParallelSchedulingGroup with a string that is a unique name for the group. The submit description file for a parallel universe job which must not cross group boundaries contains

```
+WantParallelSchedulingGroups = True
```

The dedicated scheduler enforces the allocation to within a group.

Preemption with Dedicated Jobs

The dedicated scheduler can optionally preempt running MPI jobs in favor of higher priority MPI jobs in its queue. Note that this is different from preemption in non-parallel universes, and MPI jobs cannot be preempted either by a machine's user pressing a key or by other means.

By default, the dedicated scheduler will never preempt running MPI jobs. Two configuration file items control dedicated preemption: SCHEDD_PREEMPTION_REQUIREMENTS and

SCHEDD_PREEMPTION_RANK . These have no default value, so if either are not defined, preemption will never occur. SCHEDD_PREEMPTION_REQUIREMENTS must evaluate to True for a machine to be a candidate for this kind of preemption. If more machines are candidates for preemption than needed to satisfy a higher priority job, the machines are sorted by SCHEDD_PREEMPTION_RANK, and only the highest ranked machines are taken.

Note that preempting one node of a running MPI job requires killing the entire job on all of its nodes. So, when preemption happens, it may end up freeing more machines than strictly speaking are needed. Also, as Condor cannot produce checkpoints for MPI jobs, preempted jobs will be re-run, starting again from the beginning. Thus, the administrator should be careful when enabling dedicated preemption. The following example shows how to enable dedicated preemption.

```
STARTD_JOB_EXPRS = JobPrio
SCHEDD_PREEMPTION_REQUIREMENTS = (My.JobPrio < Target.JobPrio)
SCHEDD_PREEMPTION_RANK = 0.0
```

In this case, preemption is enabled by the user job priority. If a set of machines is running a job at user priority 5, and the user submits a new job at user priority 10, the running job will be preempted for the new job. The old job is put back in the queue, and will begin again from the beginning when assigned to a new set of machines.

Grouping dedicated nodes into parallel scheduling groups

In some parallel environments, machines are divided into groups, and jobs should not cross groups of machines – that is, all the nodes of a parallel job should be allocated to machines in the same group. The most common example is a pool of machine using infiniband switches. Each switch might connect 16 machines, and a pool might have 160 machines on 10 switches. If the infiniband switches are not routed to each other, each job must run on machines connected to the same switch. The dedicated scheduler's parallel scheduling groups features supports this operation.

Each *condor_startd* must define which group it belongs to by setting the ParallelSchedulingGroup variable in the configuration file, and advertising it into the machine ClassAd. The value of this variable is a string, which should be the same for all *condor_startd* daemons in a given group. The property must be advertised in the *condor_startd* ClassAd by appending ParallelSchedulingGroup to the STARTD_ATTRS configuration variable. Then, parallel jobs which want to be scheduled by group declare this by setting +WantParallelSchedulingGroups = True in their submit description file.

3.13.11 Configuring Condor for Running Backfill Jobs

Condor can be configured to run backfill jobs whenever the *condor_startd* has no other work to perform. These jobs are considered the lowest possible priority, but when machines would otherwise be idle, the resources can be put to good use.

Currently, Condor only supports using the Berkeley Open Infrastructure for Network Computing (BOINC) to provide the backfill jobs. More information about BOINC is available at http://boinc.berkeley.edu.

The rest of this section provides an overview of how backfill jobs work in Condor, details for configuring the policy for when backfill jobs are started or killed, and details on how to configure Condor to spawn the BOINC client to perform the work.

Overview of Backfill jobs in Condor

Whenever a resource controlled by Condor is in the Unclaimed/Idle state, it is totally idle; neither the interactive user nor a Condor job is performing any work. Machines in this state can be configured to enter the *Backfill* state, which allows the resource to attempt a background computation to keep itself busy until other work arrives (either a user returning to use the machine interactively, or a normal Condor job). Once a resource enters the Backfill state, the *condor_startd* will attempt to spawn another program, called a *backfill client*, to launch and manage the backfill computation. When other work arrives, the *condor_startd* will kill the backfill client and clean up any processes it has spawned, freeing the machine resources for the new, higher priority task. More details about the different states a Condor resource can enter and all of the possible transitions between them are described in section 3.5 beginning on page 284, especially sections 3.5.5, 3.5.6, and 3.5.7.

At this point, the only backfill system supported by Condor is BOINC. The *condor_startd* has the ability to start and stop the BOINC client program at the appropriate times, but otherwise provides no additional services to configure the BOINC computations themselves. Future versions of Condor might provide additional functionality to make it easier to manage BOINC computations from within Condor. For now, the BOINC client must be manually installed and configured outside of Condor on each backfill-enabled machine.

Defining the Backfill Policy

There are a small set of policy expressions that determine if a *condor_startd* will attempt to spawn a backfill client at all, and if so, to control the transitions in to and out of the Backfill state. This section briefly lists these expressions. More detail can be found in section 3.3.10 on page 195.

ENABLE_BACKFILL A boolean value to determine if any backfill functionality should be used. The default value is `False`.

BACKFILL_SYSTEM A string that defines what backfill system to use for spawning and managing backfill computations. Currently, the only supported string is `"BOINC"`.

START_BACKFILL A boolean expression to control if a Condor resource should start a backfill client. This expression is only evaluated when the machine is in the Unclaimed/Idle state and the `ENABLE_BACKFILL` expression is `True`.

EVICT_BACKFILL A boolean expression that is evaluated whenever a Condor resource is in the Backfill state. A value of `True` indicates the machine should immediately kill the currently running backfill client and any other spawned processes, and return to the Owner state.

The following example shows a possible configuration to enable backfill:

```
# Turn on backfill functionality, and use BOINC
ENABLE_BACKFILL = TRUE
BACKFILL_SYSTEM = BOINC

# Spawn a backfill job if we've been Unclaimed for more than 5
# minutes
START_BACKFILL = $(StateTimer) > (5 * $(MINUTE))

# Evict a backfill job if the machine is busy (based on keyboard
# activity or cpu load)
EVICT_BACKFILL = $(MachineBusy)
```

Overview of the BOINC system

The BOINC system is a distributed computing environment for solving large scale scientific problems. A detailed explanation of this system is beyond the scope of this manual. Thorough documentation about BOINC is available at their website: http://boinc.berkeley.edu. However, a brief overview is provided here for sites interested in using BOINC with Condor to manage backfill jobs.

BOINC grew out of the relatively famous SETI@home computation, where volunteers installed special client software, in the form of a screen saver, that contacted a centralized server to download work units. Each work unit contained a set of radio telescope data and the computation tried to find patterns in the data, a sign of intelligent life elsewhere in the universe (hence the name: "Search for Extra Terrestrial Intelligence at home"). BOINC is developed by the Space Sciences Lab at the University of California, Berkeley, by the same people who created SETI@home. However, instead of being tied to the specific radio telescope application, BOINC is a generic infrastructure by which many different kinds of scientific computations can be solved. The current generation of SETI@home now runs on top of BOINC, along with various physics, biology, climatology, and other applications.

The basic computational model for BOINC and the original SETI@home is the same: volunteers install BOINC client software which runs whenever the machine would otherwise be idle. However, the BOINC installation on any given machine must be configured so that it knows what computations to work for (each computation is referred to as a *project* using BOINC's terminology), instead of always working on a hard coded computation. A given BOINC client can be configured to donate all of its cycles to a single project, or to split the cycles between projects so that, on average, the desired percentage of the computational power is allocated to each project. Once the client software (a program called the *boinc_client*) starts running, it attempts to contact a centralized server for each project it has been configured to work for. The BOINC software downloads the appropriate platform-specific application binary and some work units from the central server for each project. Whenever the client software completes a given work unit, it once again attempts to connect to that project's central server to upload the results and download more work.

BOINC participants must register at the centralized server for each project they wish to donate cycles to. The process produces a unique identifier so that the work performed by a given client can be credited to a specific user. BOINC keeps track of the work units completed by each user, so that users providing the most cycles get the highest rankings (and therefore, bragging rights).

Because BOINC already handles the problems of distributing the application binaries for each scientific computation, the work units, and compiling the results, it is a perfect system for managing backfill computations in Condor. Many of the applications that run on top of BOINC produce their own application-specific checkpoints, so even if the *boinc_client* is killed (for example, when a Condor job arrives at a machine, or if the interactive user returns) an entire work unit will not necessarily be lost.

Installing the BOINC client software

If a working installation of BOINC currently exists on machines where backfill is desired, skip the remainder of this section. Continue reading with the section titled "Configuring the BOINC client under Condor".

In Condor Version 7.6.0, the BOINC client software that actually spawns and manages the backfill computations (the *boinc_client*) must be manually downloaded, installed and configured outside of Condor. Hopefully in future versions, the Condor package will include the *boinc_client*, and there will be a way to automatically install and configure the BOINC software together with Condor.

The *boinc_client* executables can be obtained at one of the following locations:

http://boinc.berkeley.edu/download.php This is the official BOINC download site, which provides binaries for MacOS 10.3 or higher, Linux/x86, and Windows/x86. From the download table, use the "Recommended version", and use the "Core client only (command-line)" package when available.

http://boinc.berkeley.edu/download_other.php This page contains links to sites that distribute *boinc_client* binaries for other platforms beyond the officially supported ones.

Once the BOINC client software has been downloaded, the *boinc_client* binary should be placed in a location where the Condor daemons can use it. The path will be specified via a Condor configuration setting, `BOINC_Executable` , described below.

Additionally, a local directory on each machine should be created where the BOINC system can write files it needs. This directory must not be shared by multiple instances of the BOINC software, just like the `spool` or `execute` directories used by Condor. This location of this directory is defined using the `BOINC_InitialDir` macro, described below. The directory must be writable by whatever user the *boinc_client* will run as. This user is either the same as the user the Condor daemons are running as (if Condor is not running as root), or a user defined via the `BOINC_Owner` setting described below.

Finally, Condor administrators wishing to use BOINC for backfill jobs must create accounts at the various BOINC projects they want to donate cycles to. The details of this process vary

from project to project. Beware that this step must be done manually, as the BOINC software spawned by Condor (the *boinc_client*) can not automatically register a user at a given project (unlike the more fancy GUI version of the BOINC client software which many users run as a screen saver). For example, to configure machines to perform work for the Einstein@home project (a physics experiment run by the University of Wisconsin at Milwaukee) Condor administrators should go to http://einstein.phys.uwm.edu/create_account_form.php, fill in the web form, and generate a new Einstein@home identity. This identity takes the form of a project URL (such as http://einstein.phys.uwm.edu) followed by an *account key*, which is a long string of letters and numbers that is used as a unique identifier. This URL and account key will be needed when configuring Condor to use BOINC for backfill computations (described in the next section).

Configuring the BOINC client under Condor

This section assumes that the BOINC client software has already been installed on a given machine, that the BOINC projects to join have been selected, and that a unique project account key has been created for each project. If any of these steps has not been completed, please read the previous section titled "Installing the BOINC client software"

Whenever the *condor_startd* decides to spawn the *boinc_client* to perform backfill computations (when `ENABLE_BACKFILL` is `True`, when the resource is in Unclaimed/Idle, and when the `START_BACKFILL` expression evaluates to `True`), it will spawn a *condor_starter* to directly launch and monitor the *boinc_client* program. This *condor_starter* is just like the one used to spawn normal Condor jobs. In fact, the argv[0] of the *boinc_client* will be renamed to "condor_exec", as described in section 2.15.1 on page 121.

The *condor_starter* for spawning the *boinc_client* reads values out of the Condor configuration files to define the job it should run, as opposed to getting these values from a job classified ad in the case of a normal Condor job. All of the configuration settings to control things like the path to the *boinc_client* binary to use, the command-line arguments, the initial working directory, and so on, are prefixed with the string `"BOINC_"`. Each possible setting is described below:

Required settings:

`BOINC_Executable` The full path to the *boinc_client* binary to use.

`BOINC_InitialDir` The full path to the local directory where BOINC should run.

`BOINC_Universe` The Condor universe used for running the *boinc_client* program. This **must** be set to `"vanilla"` for BOINC to work under Condor.

`BOINC_Owner` What user the *boinc_client* program should be run as. This macro is only used if the Condor daemons are running as root. In this case, the *condor_starter* must be told what user identity to switch to before spawning the *boinc_client*. This can be any valid user on the local system, but it must have write permission in whatever directory is specified in `BOINC_InitialDir`).

Optional settings:

BOINC_Arguments Command-line arguments that should be passed to the *boinc_client* program. For example, one way to specify the BOINC project to join is to use the **–attach_project** argument to specify a project URL and account key. For example:

```
BOINC_Arguments = --attach_project http://einstein.phys.uwm.edu [account_key]
```

BOINC_Environment Environment variables that should be set for the *boinc_client*.

BOINC_Output Full path to the file where STDOUT from the *boinc_client* should be written. If this macro is not defined, STDOUT will be discarded.

BOINC_Error Full path to the file where STDERR from the *boinc_client* should be written. If this macro is not defined, STDERR will be discarded.

The following example shows one possible usage of these settings:

```
# Define a shared macro that can be used to define other settings.
# This directory must be manually created before attempting to run
# any backfill jobs.
BOINC_HOME = $(LOCAL_DIR)/boinc

# Path to the boinc_client to use, and required universe setting
BOINC_Executable = /usr/local/bin/boinc_client
BOINC_Universe = vanilla

# What initial working directory should BOINC use?
BOINC_InitialDir = $(BOINC_HOME)

# Save STDOUT and STDERR
BOINC_Output = $(BOINC_HOME)/boinc.out
BOINC_Error = $(BOINC_HOME)/boinc.err
```

If the Condor daemons reading this configuration are running as root, an additional macro must be defined:

```
# Specify the user that the boinc_client should run as:
BOINC_Owner = nobody
```

In this case, Condor would spawn the *boinc_client* as "**nobody**", so the directory specified in `$(BOINC_HOME)` would have to be writable by the "**nobody**" user.

A better choice would probably be to create a separate user account just for running BOINC jobs, so that the local BOINC installation is not writable by other processes running as "**nobody**". Alternatively, the `BOINC_Owner` could be set to "**daemon**".

Attaching to a specific BOINC project

There are a few ways to attach a Condor/BOINC installation to a given BOINC project:

- The **–attach_project** argument to the *boinc_client* program, defined via the `BOINC_Arguments` setting (described above). The *boinc_client* will only accept a single **–attach_project** argument, so this method can only be used to attach to one project.
- The *boinc_cmd* command-line tool can perform various BOINC administrative tasks, including attaching to a BOINC project. Using *boinc_cmd*, the appropriate argument to use is called **–project_attach**. Unfortunately, the *boinc_client* must be running for *boinc_cmd* to work, so this method can only be used once the Condor resource has entered the Backfill state and has spawned the *boinc_client*.
- Manually create account files in the local BOINC directory. Upon startup, the *boinc_client* will scan its local directory (the directory specified with `BOINC_InitialDir`) for files of the form `account_[URL].xml`, for example, `account_einstein.phys.uwm.edu.xml`. Any files with a name that matches this convention will be read and processed. The contents of the file define the project URL and the authentication key. The format is:

  ```
  <account>
    <master_url>[URL]</master_url>
    <authenticator>[key]</authenticator>
  </account>
  ```

  For example:

  ```
  <account>
    <master_url>http://einstein.phys.uwm.edu</master_url>
    <authenticator>aaaa1111bbbb2222cccc3333</authenticator>
  </account>
  ```

  (Of course, the `<authenticator>` tag would use the real authentication key returned when the account was created at a given project).

  These account files can be copied to the local BOINC directory on all machines in a Condor pool, so administrators can either distribute them manually, or use symbolic links to point to a shared file system.

In the first two cases (using command-line arguments for *boinc_client* or running the *boinc_cmd* tool), BOINC will write out the resulting account file to the local BOINC directory on the machine, and then future invocations of the *boinc_client* will already be attached to the appropriate project(s). More information about participating in multiple BOINC projects can be found at http://boinc.berkeley.edu/multiple_projects.php.

BOINC on Windows

The Windows version of BOINC has multiple installation methods. The preferred method of installation for use with Condor is the “Shared Installation” method. Using this method gives all users access to the executables. During the installation process

1. Deselect the option which makes BOINC the default screen saver
2. Deselect the option which runs BOINC on start-up.
3. Do not launch BOINC at the conclusion of the installation.

There are three major differences from the Unix version to keep in mind when dealing with the Windows installation:

1. The Windows executables have different names from the Unix versions. The Windows client is called *boinc.exe*. Therefore, the configuration variable `BOINC_Executable` is written:

   ```
   BOINC_Executable = C:\PROGRA~1\BOINC\boinc.exe
   ```

   The Unix administrative tool *boinc_cmd* is called *boinccmd.exe* on Windows.

2. When using BOINC on Windows, the configuration variable `BOINC_InitialDir` will not be respected fully. To work around this difficulty, pass the BOINC home directory directly to the BOINC application via the `BOINC_Arguments` configuration variable. For Windows, rewrite the argument line as:

   ```
   BOINC_Arguments = --dir $(BOINC_HOME) \
             --attach_project http://einstein.phys.uwm.edu [account_key]
   ```

   As a consequence of setting the BOINC home directory, some projects may fail with the authentication error:

   ```
   Scheduler request failed: Peer
   certificate cannot be authenticated
   with known CA certificates.
   ```

   To resolve this issue, copy the `ca-bundle.crt` file from the BOINC installation directory to `$(BOINC_HOME)`. This file appears to be project and machine independent, and it can therefore be distributed as part of an automated Condor installation.

3. The `BOINC_Owner` configuration variable behaves differently on Windows than it does on Unix. Its value may take one of two forms:

   - `domain\user`
   - `user`
     This form assumes that the user exists in the local domain (that is, on the computer itself).

   Setting this option causes the addition of the job attribute

   ```
   RunAsUser = True
   ```

   to the backfill client. This further implies that the configuration variable `STARTER_ALLOW_RUNAS_OWNER` be set to `True` to insure that the local *condor_starter* be able to run jobs in this manner. For more information on the `RunAsUser` attribute, see section 6.2.4. For more information on the the `STARTER_ALLOW_RUNAS_OWNER` configuration variable, see section 3.3.7.

3.13.12 Group ID-Based Process Tracking

One function that Condor often must perform is keeping track of all processes created by a job. This is done so that Condor can provide resource usage statistics about jobs, and also so that Condor can properly clean up any processes that jobs leave behind when they exit.

In general, tracking process families is difficult to do reliably. By default Condor uses a combination of process parent-child relationships, process groups, and information that Condor places in a job's environment to track process families on a best-effort basis. This usually works well, but it can falter for certain applications or for jobs that try to evade detection.

Jobs that run with a user account dedicated for Condor's use can be reliably tracked, since all Condor needs to do is look for all processes running using the given account. Administrators must specify in Condor's configuration what accounts can be considered dedicated via the `DEDICATED_EXECUTE_ACCOUNT_REGEXP` setting. See Section 3.6.13 for further details.

Ideally, jobs can be reliably tracked regardless of the user account they execute under. This can be accomplished with group ID-based tracking. This method of tracking requires that a range of dedicated *group* IDs (GID) be set aside for Condor's use. The number of GIDs that must be set aside for an execute machine is equal to its number of execution slots. GID-based tracking is only available on Linux, and it requires that Condor either runs as `root` or uses privilege separation (see Section 3.6.14).

GID-based tracking works by placing a dedicated GID in the supplementary group list of a job's initial process. Since modifying the supplementary group ID list requires `root` privilege, the job will not be able to create processes that go unnoticed by Condor.

Once a suitable GID range has been set aside for process tracking, GID-based tracking can be enabled via the `USE_GID_PROCESS_TRACKING` parameter. The minimum and maximum GIDs included in the range are specified with the `MIN_TRACKING_GID` and `MAX_TRACKING_GID` settings. For example, the following would enable GID-based tracking for an execute machine with 8 slots.

```
USE_GID_PROCESS_TRACKING = True
MIN_TRACKING_GID = 750
MAX_TRACKING_GID = 757
```

If the defined range is too small, such that there is not a GID available when starting a job, then the *condor_starter* will fail as it tries to start the job. An error message will be logged stating that there are no more tracking GIDs.

GID-based process tracking requires use of the *condor_procd*. If `USE_GID_PROCESS_TRACKING` is true, the *condor_procd* will be used regardless of the `USE_PROCD` setting. Changes to `MIN_TRACKING_GID` and `MAX_TRACKING_GID` require a full restart of Condor.

3.13.13 Limiting Resource Usage

An administrator can strictly limit the usage of system resources by jobs for any job that may be wrapped using the script defined by the configuration variable USER_JOB_WRAPPER . These are jobs within universes that are controlled by the *condor_starter* daemon, and they include the **vanilla**, **standard**, **java**, **local**, and **parallel** universes.

The job's ClassAd is written by the *condor_starter* daemon. It will need to contain attributes that the script defined by USER_JOB_WRAPPER can use to implement platform specific resource limiting actions. Examples of resources that may be referred to for limiting purposes are RAM, swap space, file descriptors, stack size, and core file size.

An initial sample of a USER_JOB_WRAPPER script is provided in the installation at $(LIBEXEC)/condor_limits_wrapper.sh. Here is the contents of that file:

```
#!/bin/sh
# Copyright 2008 Red Hat, Inc.
#
# Licensed under the Apache License, Version 2.0 (the "License");
# you may not use this file except in compliance with the License.
# You may obtain a copy of the License at
#
#     http://www.apache.org/licenses/LICENSE-2.0
#
# Unless required by applicable law or agreed to in writing, software
# distributed under the License is distributed on an "AS IS" BASIS,
# WITHOUT WARRANTIES OR CONDITIONS OF ANY KIND, either express or implied.
# See the License for the specific language governing permissions and
# limitations under the License.

if [[ $_CONDOR_MACHINE_AD != "" ]]; then
   mem_limit=$((`egrep '^Memory' $_CONDOR_MACHINE_AD | cut -d ' ' -f 3` * 1024))
#   block_size=$((`stat -f -c %s .` / 1024))
#   disk_limit=$((`egrep '^Disk' $_CONDOR_MACHINE_AD | cut -d ' ' -f 3` / $block_size))
   disk_limit=`egrep '^Disk' $_CONDOR_MACHINE_AD | cut -d ' ' -f 3`
   vm_limit=`egrep '^VirtualMemory' $_CONDOR_MACHINE_AD | cut -d ' ' -f 3`

   ulimit -d $mem_limit
   if [[ $? != 0 ]] || [[ $mem_limit = "" ]]; then
      echo "Failed to set Memory Resource Limit" > $_CONDOR_WRAPPER_ERROR_FILE
      exit 1
   fi
   ulimit -f $disk_limit
   if [[ $? != 0 ]] || [[ $disk_limit = "" ]]; then
      echo "Failed to set Disk Resource Limit" > $_CONDOR_WRAPPER_ERROR_FILE
      exit 1
   fi
   ulimit -v $vm_limit
   if [[ $? != 0 ]] || [[ $vm_limit = "" ]]; then
      echo "Failed to set Virtual Memory Resource Limit" > $_CONDOR_WRAPPER_ERROR_FILE
      exit 1
   fi
fi
```

```
exec "$@"
error=$?
echo "Failed to exec($error): $@" > $_CONDOR_WRAPPER_ERROR_FILE
exit 1
```

If used in an unmodified form, this script sets the job's limits on a per slot basis for memory, disk, and virtual memory usage, with the limits defined by the values in the machine ClassAd. This example file will need to be modified and merged for use with a preexisting USER_JOB_WRAPPER script.

If additional functionality is added to the script, an administrator is likely to use the USER_JOB_WRAPPER script in conjunction with SUBMIT_EXPRS to force the job ClassAd to contain attributes that the USER_JOB_WRAPPER script expects to have defined.

The following variables are set in the environment of the the USER_JOB_WRAPPER script by the *condor_starter* daemon, when the USER_JOB_WRAPPER is defined.

_CONDOR_MACHINE_AD The full path and file name of the file containing the machine ClassAd.

_CONDOR_JOB_AD The full path and file name of the file containing the job ClassAd.

_CONDOR_WRAPPER_ERROR_FILE The full path and file name of the file that the USER_JOB_WRAPPER script should create, if there is an error. The text in this file will be included in any Condor failure messages.

3.13.14 Concurrency Limits

Condor's implementation of the mechanism called *concurrency limits* allows an administrator to define and set integer limits on consumable resources. These limits are utilized during matchmaking, preventing matches when the resources are allocated. Typical uses of this mechanism will include the management of software licenses, database connections, and any other consumable resource external to Condor.

Use of the concurrency limits mechanism requires configuration variables to set distinct limits, while jobs must identify the need for a specific resource.

In the configuration, a string must be chosen as a name for the particular resource. This name is used in the configuration of a *condor_negotiator* daemon variable that defines the concurrency limit, or integer quantity available of this resource. For example, assume that there are 3 licenses for the X software. The configuration variable concurrency limit may be:

```
XSW_LIMIT = 3
```

where "XSW" is the invented name of this resource, which is appended with the string _LIMIT. With this limit, a maximum of 3 jobs declaring that they need this resource may be executed concurrently.

In addition to named limits, such as in the example named limit XSW, configuration may specify a concurrency limit for all resources that are not covered by specifically-named limits. The configuration variable CONCURRENCY_LIMIT_DEFAULT sets this value. For example,

```
CONCURRENCY_LIMIT_DEFAULT = 1
```

sets a limit of 1 job in execution for any job that declares its requirement for a resource that is not named in the configuration. If CONCURRENCY_LIMIT_DEFAULT is omitted from the configuration, then no limits are placed on the number of concurrently executing jobs of resources for which there is no specifically named concurrency limit.

The job must declare its need for a resource by placing a command in its submit description file or adding an attribute to the job ClassAd. In the submit description file, an example job that requires the X software adds:

```
concurrency_limits = XSW
```

This results in the job ClassAd attribute

```
ConcurrencyLimits = "XSW"
```

The implementation of the job ClassAd attribute ConcurrencyLimits has a more general implementation. It is either a string or a string list. A list contains items delimited by space characters and comma characters. Therefore, a job that requires the 3 separate resources named as "XSW", "y", and "Z", will contain in its submit description file:

```
concurrency_limits = y,XSW,Z
```

Additionally, a numerical value identifying the number of resources required may be specified in the definition of a resource, following the resource name by a colon character and the integer number of resources. Modifying the given example to specify that 3 of the "XSW" resource are needed results in:

```
concurrency_limits = y,XSW:3,Z
```

Note that the maximum for any given limit, as specified with the configuration variable <*>_LIMIT, is as strictly enforced **as possible**. In the presence of preemption and dropped updates from the *condor_startd* daemon to the *condor_collector* daemon, it is possible for the limit to be exceeded. Condor will never kill a job to free up a limit, including the case where a limit maximum is exceeded.

3.14 Java Support Installation

Compiled Java programs may be executed (under Condor) on any execution site with a Java Virtual Machine (JVM). To do this, Condor must be informed of some details of the JVM installation.

Begin by installing a Java distribution according to the vendor's instructions. We have successfully used the Sun Java Developer's Kit, but any distribution should suffice. Your machine may have been delivered with a JVM already installed – installed code is frequently found in `/usr/bin/java`.

Condor's configuration includes the location of the installed JVM. Edit the configuration file. Modify the `JAVA` entry to point to the JVM binary, typically `/usr/bin/java`. Restart the *condor_startd* daemon on that host. For example,

```
% condor_restart -startd bluejay
```

The *condor_startd* daemon takes a few moments to exercise the Java capabilities of the *condor_starter*, query its properties, and then advertise the machine to the pool as Java-capable. If the set up succeeded, then *condor_status* will tell you the host is now Java-capable by printing the Java vendor and the version number:

```
% condor_status -java bluejay
```

After a suitable amount of time, if this command does not give any output, then the *condor_starter* is having difficulty executing the JVM. The exact cause of the problem depends on the details of the JVM, the local installation, and a variety of other factors. We can offer only limited advice on these matters, but here is an approach to solving the problem.

To reproduce the test that the *condor_starter* is attempting, try running the Java *condor_starter* directly. To find where the *condor_starter* is installed, run this command:

```
% condor_config_val STARTER
```

This command prints out the path to the *condor_starter*, perhaps something like this:

```
/usr/condor/sbin/condor_starter
```

Use this path to execute the *condor_starter* directly with the *-classad* argument. This tells the starter to run its tests and display its properties.

```
/usr/condor/sbin/condor_starter -classad
```

This command will display a short list of cryptic properties, such as:

```
IsDaemonCore = True
HasFileTransfer = True
HasMPI = True
CondorVersion = "$CondorVersion: 7.1.0 Mar 26 2008 BuildID: 80210 $"
```

If the Java configuration is correct, there will also be a short list of Java properties, such as:

```
JavaVendor = "Sun Microsystems Inc."
JavaVersion = "1.2.2"
JavaMFlops = 9.279696
HasJava = True
```

If the Java installation is incorrect, then any error messages from the shell or Java will be printed on the error stream instead.

The Sun JVM sets a value of 64 Mbytes for the Java Maxheap Argument, which Condor uses. This value is often too small for the application. The administrator can change this value through configuration by setting a different value for JAVA_EXTRA_ARGUMENTS .

```
JAVA_EXTRA_ARGUMENTS = -Xmx1024m
```

Note that if a specific job sets the value in the submit description file, using the submit command **java_vm_args**, this job's value takes precedence over a configured value.

3.15 Virtual Machines

Virtual machines can be executed on any execution site with VMware, Xen (via *libvirt*), or KVM. To do this, Condor must be informed of some details of the virtual machine installation, and the execution machines must be configured correctly. This permits the execution of **vm** universe jobs.

What follows is not a comprehensive list of the options that help set up to use the **vm** universe; rather, it is intended to serve as a starting point for those users interested in getting **vm** universe jobs up and running quickly. Details of configuration variables are in section 3.3.29.

Begin by installing the virtualization package on all execute machines, according to the vendor's instructions. We have successfully used VMware Server, Xen, and KVM. If considering running on a Windows system, a *Perl* distribution will also need to be installed; we have successfully used *ActivePerl*.

For VMware, *VMware Server 1* must be installed and running on the execute machine.

For Xen, there are three things that must exist on an execute machine to fully support **vm** universe jobs.

1. A Xen-enabled kernel must be running. This running Xen kernel acts as Dom0, in Xen terminology, under which all VMs are started, called DomUs Xen terminology.
2. The *libvirtd* daemon must be available, and *Xend* services must be running.
3. The *pygrub* program must be available, for execution of VMs whose disks contain the kernel they will run.

For KVM, there are two things that must exist on an execute machine to fully support **vm** universe jobs.

1. The machine must have the KVM kernel module installed and running.
2. The *libvirtd* daemon must be installed and running.

3.15.1 Configuration Variables

There are configuration variables related to the virtual machines for **vm** universe jobs. Some options are required, while others are optional. Here we only discuss those that are required.

First, the type of virtual machine that is installed on the execute machine must be specified. For now, only one type can be utilized per machine. For instance, the following tells Condor to use VMware:

```
VM_TYPE = vmware
```

The location of the *condor_vm-gahp* and its log file must also be specified on the execute machine. On a Windows installation, these options would look like this:

```
VM_GAHP_SERVER = $(SBIN)/condor_vm-gahp.exe
VM_GAHP_LOG = $(LOG)/VMGahpLog
```

VMware-Specific Configuration

To use VMware, identify the location of the *Perl* executable on the execute machine. In most cases, the default value should suffice:

```
VMWARE_PERL = perl
```

This, of course, assumes the *Perl* executable is in the path of the *condor_master* daemon. If this is not the case, then a full path to the *Perl* executable will be required.

The final required configuration is the location of the VMware control script used by the *condor_vm-gahp* on the execute machine to talk to the virtual machine hypervisor. It is located in Condor's `sbin` directory:

```
VMWARE_SCRIPT = $(SBIN)/condor_vm_vmware.pl
```

Note that an execute machine's EXECUTE variable should not contain any symbolic links in its path, if the machine is configured to run VMware **vm** universe jobs. See the FAQ entry in section 7.3 for details.

Xen-Specific and KVM-Specific Configuration

Once the configuration options have been set, restart the *condor_startd* daemon on that host. For example:

```
> condor_restart -startd leovinus
```

The *condor_startd* daemon takes a few moments to exercise the VM capabilities of the *condor_vm-gahp*, query its properties, and then advertise the machine to the pool as VM-capable. If the set up succeeded, then *condor_status* will reveal that the host is now VM-capable by printing the VM type and the version number:

```
> condor_status -vm leovinus
```

After a suitable amount of time, if this command does not give any output, then the *condor_vm-gahp* is having difficulty executing the VM software. The exact cause of the problem depends on the details of the VM, the local installation, and a variety of other factors. We can offer only limited advice on these matters:

For Xen and KVM, the **vm** universe is only available when `root` starts Condor. This is a restriction currently imposed because root privileges are required to create a virtual machine on top of a Xen-enabled kernel. Specifically, root is needed to properly use the *libvirt* utility that controls creation and management of Xen and KVM guest virtual machines. This restriction may be lifted in future versions, depending on features provided by the underlying tool *libvirt*.

3.16 Power Management

Condor supports placing machines in low power states. Power setting decisions are based upon Condor configuration.

Power conservation is relevant when machines are not in heavy use, or when there are known periods of low activity within the pool.

3.16.1 Entering a Low Power State

By default, Condor does not do power management. When desired, the ability to place a machine into a low power state is accomplished through configuration. This occurs when all slots on a machine agree that a low power state is desired.

A slot's readiness to hibernate is determined by the evaluating the HIBERNATE configuration variable (see section 3.3.10 on page 205) within the context of the slot. Readiness is evaluated at fixed intervals, as determined by the HIBERNATE_CHECK_INTERVAL configuration variable. A non-zero value of this variable enables the power management facility. It is an integer value representing seconds, and it need not be a small value. There is a trade off between the extra time not at a low power state and the unnecessary computation of readiness.

To put the machine in a low power state rapidly after it has become idle, consider checking each slot's state frequently, as in the example configuration:

```
HIBERNATE_CHECK_INTERVAL = 20
```

This checks each slot's readiness every 20 seconds. A more common value for frequency of checks is 300 (5 minutes). A value of 300 loses some degree of granularity, but it is more reasonable as machines are likely to be put in to a low power state after a few hours, rather than minutes.

A slot's readiness or willingness to enter a low power state is determined by the HIBERNATE expression. Because this expression is evaluated in the context of each slot, and not on the machine as a whole, any one slot can veto a change of power state. The HIBERNATE expression may reference a wide array of variables. Possibilities include the change in power state if none of the slots are claimed, or if the slots are not in the Owner state.

Here is a concrete example. Assume that the START expression is not set to always be True. This permits an easy determination whether or not the machine is in an Unclaimed state through the use of an auxiliary macro called ShouldHibernate.

```
TimeToWait  = (2 * $(HOUR))
ShouldHibernate = ( (KeyboardIdle > $(StartIdleTime)) \
                    && $(CPUIdle) \
                    && ($(StateTimer) > $(TimeToWait)) )
```

This macro evaluates to True if the following are all True:

- The keyboard has been idle long enough.
- The CPU is idle.
- The slot has been Unclaimed for more than 2 hours.

The sample HIBERNATE expression that enters the power state called "RAM", if ShouldHibernate evaluates to True, and remains in its current state otherwise is

```
HibernateState  = "RAM"
HIBERNATE = ifThenElse($(ShouldHibernate), $(HibernateState), "NONE" )
```

If any slot returns "NONE", that slot vetoes the decision to enter a low power state. Only when values returned by all slots are all non-zero is there a decision to enter a low power state. If all agree to enter the low power state, but differ in which state to enter, then the largest magnitude value is chosen.

3.16.2 Returning From a Low Power State

The Condor command line tool *condor_power* may wake a machine from a low power state by sending a UDP Wake On LAN (WOL) packet. See the *condor_power* manual page on page 763.

To automatically call *condor_power* under specific conditions, *condor_rooster* may be used. The configuration options for *condor_rooster* are described in section 3.3.35.

3.16.3 Keeping a ClassAd for a Hibernating Machine

A pool's *condor_collector* daemon can be configured to keep a persistent ClassAd entry for each machine, once it has entered hibernation. This is required by *condor_rooster* so that it can evaluate the UNHIBERNATE expression of the offline machines.

To do this, define a log file using the OFFLINE_LOG configuration variable. See section 3.3.10 on page 206 for the definition. An optional expiration time for each ClassAd can be specified with OFFLINE_EXPIRE_ADS_AFTER . The timing begins from the time the hibernating machine's ClassAd enters the *condor_collector* daemon. See section 3.3.10 on page 207 for the definition.

3.16.4 Linux Platform Details

Depending on the Linux distribution and version, there are three methods for controlling a machine's power state. The methods:

1. *pm-utils* is a set of command line tools which can be used to detect and switch power states. In Condor, this is defined by the string "pm-utils".
2. The directory in the virtual file system /sys/power contains virtual files that can be used to detect and set the power states. In Condor, this is defined by the string "/sys".
3. The directory in the virtual file system /proc/acpi contains virtual files that can be used to detect and set the power states. In Condor, this is defined by the string "/proc".

By default, the Condor attempts to detect the method to use in the order shown. The first method detected as usable on the system is chosen.

This ordered detection may be bypassed, to use a specified method instead by setting the configuration variable LINUX_HIBERNATION_METHOD with one of the defined strings. This variable is defined in section 3.3.10 on page 206. If no usable methods are detected or the method specified by LINUX_HIBERNATION_METHOD is either not detected or invalid, hibernation is disabled.

The details of this selection process, and the final method selected can be logged via enabling D_FULLDEBUG in the relevant subsystem's log configuration.

3.16.5 Windows Platform Details

If after a suitable amount of time, a Windows machine has not entered the expected power state, then Condor is having difficulty exercising the operating system's low power capabilities. While the cause will be specific to the machine's hardware, it may also be due to improperly configured software. For hardware difficulties, the likely culprit is the configuration within the machine's BIOS, for which Condor can offer little guidance. For operating system difficulties, the Vista *powercfg* tool can be used to discover the available power states on the machine. The following command demonstrates how to list all of the supported power states of the machine:

```
> powercfg -A
The following sleep states are available on this system:
Standby (S3) Hibernate Hybrid Sleep
The following sleep states are not available on this system:
Standby (S1)
        The system firmware does not support this standby state.
Standby (S2)
        The system firmware does not support this standby state.
```

Note that the HIBERNATE expression is written in terms of the Sn state, where n is the value evaluated from the expression.

This tool can also be used to enable and disable other sleep states. This example turns hibernation on.

```
> powercfg -h on
```

If this tool is insufficient for configuring the machine in the manner required, the *Power Options* control panel application offers the full extent of the machine's power management abilities. Windows 2000 and XP lack the *powercfg* program, so all configuration must be done via the *Power Options* control panel application.

CHAPTER

FOUR

Miscellaneous Concepts

This chapter contains sections describing a variety of key Condor concepts that do not belong in other chapters.

ClassAds and the ClassAd language are presented.

Details of checkpoints are presented.

Description and useage of COD (Computing on Demand) extensions to Condor are presented.

The various APIs that Condor implements are described.

4.1 Condor's ClassAd Mechanism

ClassAds are a flexible mechanism for representing the characteristics and constraints of machines and jobs in the Condor system. ClassAds are used extensively in the Condor system to represent jobs, resources, submitters and other Condor daemons. An understanding of this mechanism is required to harness the full flexibility of the Condor system.

A ClassAd is is a set of uniquely named expressions. Each named expression is called an *attribute*. Figure 4.1 shows an example of a ClassAd with ten attributes.

ClassAd expressions look very much like expressions in C, and are composed of literals and attribute references composed with operators and functions. The difference between ClassAd expressions and C expressions arise from the fact that ClassAd expressions operate in a much more dynamic environment. For example, an expression from a machine's ClassAd may refer to an attribute in a job's ClassAd, such as `TARGET.Owner` in the above example. The value and type of the attribute is not known until the expression is evaluated in an environment which pairs a specific

```
MyType       = "Machine"
TargetType   = "Job"
Machine      = "froth.cs.wisc.edu"
Arch         = "INTEL"
OpSys        = "LINUX"
Disk         = 35882
Memory       = 128
KeyboardIdle = 173
LoadAvg      = 0.1000
Requirements = TARGET.Owner=="smith" || LoadAvg<=0.3 && KeyboardIdle>15*60
```

Figure 4.1: An example ClassAd

job ClassAd with the machine ClassAd.

ClassAd expressions handle these uncertainties by defining all operators to be *total* operators, which means that they have well defined behavior regardless of supplied operands. This functionality is provided through two distinguished values, `UNDEFINED` and `ERROR`, and defining all operators so that they can operate on all possible values in the ClassAd system. For example, the multiplication operator which usually only operates on numbers, has a well defined behavior if supplied with values which are not meaningful to multiply. Thus, the expression `10 * "A string"` evaluates to the value `ERROR`. Most operators are *strict* with respect to `ERROR`, which means that they evaluate to `ERROR` if any of their operands are `ERROR`. Similarly, most operators are strict with respect to `UNDEFINED`.

4.1.1 ClassAds: Old and New

ClassAds have existed for quite some time in two forms: Old and New. Old ClassAds were the original form and were used in Condor until Condor version 7.5.0. They were heavily tied to the Condor development libraries. New ClassAds added new features and were designed as a standalone library that could be used apart from Condor.

In Condor version 7.5.1, Condor switched the internal usage of ClassAds from Old to New. All user interaction with tools (such as *condor_q*) as well as output of tools is still done as Old ClassAds. Before Condor version 7.5.1, New ClassAds were used in just a few places within Condor, for example, in the Job Router and in *condor_q* **-better-analyze**. There are some syntax and behavior differences between Old and New ClassAds, all of which will remain invisible to users of Condor for this version. A complete description of New ClassAds can be found at http://www.cs.wisc.edu/condor/classad/, and in the ClassAd Language Reference Manual found on this web page.

Some of the features of New ClassAds that are *not* in Old ClassAds are lists, nested ads, time values, and matching groups of ads. Condor will avoid using these features until the 7.7.x development series, as using them makes it difficult to interact with older versions of Condor.

The syntax varies slightly between Old and New ClassAds. Here is an example ClassAd presented in both forms. The Old form:

```
Foo = 3
Bar = "ab\"cd\ef"
Moo = Foo =!= Undefined
```

The New form:

```
[
Foo = 3;
Bar = "ab\"cd\\ef";
Moo = Foo isnt Undefined;
]
```

Condor will convert to and from Old ClassAd syntax as needed.

New ClassAd Attribute References

Expressions often refer to ClassAd attributes. These attribute references work differently in Old ClassAds as compared with New ClassAds. In New ClassAds, an unscoped reference is looked for only in the local ClassAd. An *unscoped reference* is an attribute that does not have a `MY.` or `TARGET.` prefix. The *local ClassAd* may be described by an example. Matchmaking uses two ClassAds: the job ClassAd and the machine ClassAd. The job ClassAd is evaluated to see if it is a match for the machine ClassAd. The job ClassAd is the local ClassAd. Therefore, in the `Requirements` attribute of the job ClassAd, any attribute without the prefix `TARGET.` is looked up only in the job ClassAd. With New ClassAd evaluation, the use of the prefix `MY.` is eliminated, as an unscoped reference can only refer to the local ClassAd.

The `MY.` and `TARGET.` scoping prefixes only apply when evaluating an expression within the context of two ClassAds. Two examples that exemplify this are matchmaking and machine policy evaluation. When evaluating an expression within the context of a single ClassAd, `MY.` and `TARGET.` are not defined. Using them within the context of a single ClassAd will result in a value of `Undefined`. Two examples that exemplify evaluating an expression within the context of a single ClassAd are during user job policy evaluation, and with the **-constraint** option to command-line tools.

New ClassAds have no `CurrentTime` attribute. If needed, use the `time()` function instead. In order to mimic Old ClassAd semantics in this Condor version 7.5.1 release, all ClassAds have an explicit `CurrentTime` attribute, with a value of `time()`.

In this Condor version 7.5.1 release, New ClassAds will mimic the evaluation behavior of Old ClassAds. No configuration variables or submit description file contents should need to be changed. To eliminate this behavior and use only the semantics of New ClassAds, set the configuration variable `STRICT_CLASSAD_EVALUATION` to `True`. This permits testing expressions to see if any adjustment is required, before a future version of Condor potentially makes New ClassAds evaluation behavior the default or the only option.

4.1.2 Old ClassAd Syntax

ClassAd expressions are formed by composing literals, attribute references and other sub-expressions with operators and functions.

Literals

Literals in the ClassAd language may be of integer, real, string, undefined or error types. The syntax of these literals is as follows:

Integer A sequence of continuous digits (i.e., `[0-9]`). Additionally, the keywords `TRUE` and `FALSE` (case insensitive) are syntactic representations of the integers 1 and 0 respectively.

Real Two sequences of continuous digits separated by a period (i.e., `[0-9]+.[0-9]+`).

String A double quote character, followed by an list of characters terminated by a double quote character. A backslash character inside the string causes the following character to be considered as part of the string, irrespective of what that character is.

Undefined The keyword `UNDEFINED` (case insensitive) represents the `UNDEFINED` value.

Error The keyword `ERROR` (case insensitive) represents the `ERROR` value.

Attributes

Every expression in a ClassAd is named by an *attribute name*. Together, the (name,expression) pair is called an *attribute*. An attributes may be referred to in other expressions through its attribute name.

Attribute names are sequences of alphabetic characters, digits and underscores, and may not begin with a digit. All characters in the name are significant, but case is *not* significant. Thus, `Memory`, `memory` and `MeMoRy` all refer to the same attribute.

An *attribute reference* consists of the name of the attribute being referenced, and an optional *scope resolution prefix*. The prefixes that may be used are `MY.` and `TARGET.`. The case used for these prefixes is *not* significant. The semantics of supplying a prefix are discussed in Section 4.1.3.

Operators

The operators that may be used in ClassAd expressions are similar to those available in C. The available operators and their relative precedence is shown in figure 4.2. The operator with the highest precedence is the unary minus operator. The only operators which are unfamiliar are the `=?=` and `=!=` operators, which are discussed in Section 4.1.3.

```
- (unary negation)   (high precedence)
*   /
+   - (addition, subtraction)
<   <=   >=   >
==  !=  =?=  =!=
&&
||                  (low precedence)
```

Figure 4.2: Relative precedence of ClassAd expression operators

Predefined Functions

Any ClassAd expression may utilize predefined functions. Function names are case insensitive. Parameters to functions and a return value from a function may be typed (as given) or not. Nested or recursive function calls are allowed.

Here are descriptions of each of these predefined functions. The possible types are the same as itemized in in Section 4.1.2. Where the type may be any of these literal types, it is called out as `AnyType`. Where the type is `Integer`, but only returns the value 1 or 0 (implying `True` or `False`), it is called out as `Boolean`. The format of each function is given as

```
ReturnType FunctionName(ParameterType parameter1, ParameterType parameter2, ...)
```

Optional parameters are given within square brackets.

`AnyType eval(AnyType Expr)` Evaluates `Expr` as a string and then returns the result of evaluating the *contents* of the string as a ClassAd expression. This is useful when referring to an attribute such as `slotX_State` where `X`, the desired slot number is an expression, such as `SlotID+10`. In such a case, if attribute `SlotID` is 5, the value of the attribute `slot15_State` can be referenced using the expression `eval(strcat("slot", SlotID+10,"_State"))`. Function `strcat()` calls function `string()` on the second parameter, which evaluates the expression, and then converts the integer result 15 to the string "15." The concatenated string returned by `strcat()` is `"slot15_State"`, and this string is then evaluated.

Note that referring to attributes of a job from within the string passed to `eval()` in the `Requirements` or `Rank` expressions could cause inaccuracies in Condor's automatic auto-clustering of jobs into equivalent groups for matchmaking purposes. This is because Condor needs to determine which ClassAd attributes are significant for matchmaking purposes, and indirect references from within the string passed to `eval()` will not be counted.

`AnyType ifThenElse(AnyType IfExpr,AnyType ThenExpr, AnyType ElseExpr)` A conditional expression is described by `IfExpr`. The following defines return values, when `IfExpr` evaluates to

- `True`. Evaluate and return the value as given by `ThenExpr`.

- `False`. Evaluate and return the value as given by `ElseExpr`.
- `UNDEFINED`. Return the value `UNDEFINED`.
- `ERROR`. Return the value `ERROR`.
- `0.0`. Evaluate, and return the value as given by `ElseExpr`.
- non-`0.0` Real values. Evaluate, and return the value as given by `ThenExpr`.

Where `IfExpr` evaluates to give a value of type `String`, the function returns the value `ERROR`. The implementation uses lazy evaluation, so expressions are only evaluated as defined.

This function returns `ERROR` if other than exactly 3 arguments are given.

`Boolean isUndefined(AnyType Expr)` Returns `True`, if `Expr` evaluates to `UNDEFINED`. Returns `False` in all other cases.

This function returns `ERROR` if other than exactly 1 argument is given.

`Boolean isError(AnyType Expr)` Returns `True`, if `Expr` evaluates to `ERROR`. Returns `False` in all other cases.

This function returns `ERROR` if other than exactly 1 argument is given.

`Boolean isString(AnyType Expr)` Returns `True`, if the evaluation of `Expr` gives a value of type `String`. Returns `False` in all other cases.

This function returns `ERROR` if other than exactly 1 argument is given.

`Boolean isInteger(AnyType Expr)` Returns `True`, if the evaluation of `Expr` gives a value of type `Integer`. Returns `False` in all other cases.

This function returns `ERROR` if other than exactly 1 argument is given.

`Boolean isReal(AnyType Expr)` Returns `True`, if the evaluation of `Expr` gives a value of type `Real`. Returns `False` in all other cases.

This function returns `ERROR` if other than exactly 1 argument is given.

`Boolean isBoolean(AnyType Expr)` Returns `True`, if the evaluation of `Expr` gives the integer value 0 or 1. Returns `False` in all other cases.

This function returns `ERROR` if other than exactly 1 argument is given.

`Integer int(AnyType Expr)` Returns the integer value as defined by `Expr`. Where the type of the evaluated `Expr` is `Real`, the value is truncated (round towards zero) to an integer. Where the type of the evaluated `Expr` is `String`, the string is converted to an integer using a C-like `atoi()` function. When this result is not an integer, `ERROR` is returned. Where the evaluated `Expr` is `ERROR` or `UNDEFINED`, `ERROR` is returned.

This function returns `ERROR` if other than exactly 1 argument is given.

`Real real(AnyType Expr)` Returns the real value as defined by `Expr`. Where the type of the evaluated `Expr` is `Integer`, the return value is the converted integer. Where the type of the evaluated `Expr` is `String`, the string is converted to a real value using a C-like `atof()`

function. When this result is not a real, ERROR is returned. Where the evaluated Expr is ERROR or UNDEFINED, ERROR is returned.

This function returns ERROR if other than exactly 1 argument is given.

String string(AnyType Expr) Returns the string that results from the evaluation of Expr. Converts a non-string value to a string. Where the evaluated Expr is ERROR or UNDEFINED, ERROR is returned.

This function returns ERROR if other than exactly 1 argument is given.

Integer floor(AnyType Expr) Returns the integer that results from the evaluation of Expr, where the type of the evaluated Expr is Integer. Where the type of the evaluated Expr is *not* Integer, function real(Expr) is called. Its return value is then used to return the largest magnitude integer that is not larger than the returned value. Where real(Expr) returns ERROR or UNDEFINED, ERROR is returned.

This function returns ERROR if other than exactly 1 argument is given.

Integer ceiling(AnyType Expr) Returns the integer that results from the evaluation of Expr, where the type of the evaluated Expr is Integer. Where the type of the evaluated Expr is *not* Integer, function real(Expr) is called. Its return value is then used to return the smallest magnitude integer that is not less than the returned value. Where real(Expr) returns ERROR or UNDEFINED, ERROR is returned.

This function returns ERROR if other than exactly 1 argument is given.

Integer round(AnyType Expr) Returns the integer that results from the evaluation of Expr, where the type of the evaluated Expr is Integer. Where the type of the evaluated Expr is *not* Integer, function real(Expr) is called. Its return value is then used to return the integer that results from a round-to-nearest rounding method. The nearest integer value to the return value is returned, except in the case of the value at the exact midpoint between two integer values. In this case, the even valued integer is returned. Where real(Expr) returns ERROR or UNDEFINED, or the integer value does not fit into 32 bits, ERROR is returned.

This function returns ERROR if other than exactly 1 argument is given.

Integer random([AnyType Expr]) Where the optional argument Expr evaluates to type Integer or type Real (and called x), the return value is the integer or real r randomly chosen from the interval 0 <= r < x. With no argument, the return value is chosen with random(1.0). Returns ERROR in all other cases.

This function returns ERROR if greater than 1 argument is given.

String strcat(AnyType Expr1 [, AnyType Expr2 ...**])** Returns the string which is the concatenation of all arguments, where all arguments are converted to type String by function string(Expr). Returns ERROR if any argument evaluates to UNDEFINED or ERROR.

String substr(String s, Integer offset [, Integer length]) Returns the substring of s, from the position indicated by offset, with (optional) length characters. The first character within s is at offset 0. If the optional length argument is

not present, the substring extends to the end of the string. If `offset` is negative, the value `(length - offset)` is used for the offset. If `length` is negative, an initial substring is computed, from the offset to the end of the string. Then, the absolute value of `length` characters are deleted from the right end of the initial substring. Further, where characters of this resulting substring lie outside the original string, the part that lies within the original string is returned. If the substring lies completely outside of the original string, the null string is returned.

This function returns `ERROR` if greater than 3 or less than 2 arguments are given.

`Integer strcmp(AnyType Expr1, AnyType Expr2)` Both arguments are converted to type `String` by function `string(Expr)`. The return value is an integer that will be

- less than 0, if `Expr1` is lexicographically less than `Expr2`
- equal to 0, if `Expr1` is lexicographically equal to `Expr2`
- greater than 0, if `Expr1` is lexicographically greater than `Expr2`

Case is significant in the comparison. Where either argument evaluates to `ERROR` or `UNDEFINED`, `ERROR` is returned.

This function returns `ERROR` if other than 2 arguments are given.

`Integer stricmp(AnyType Expr1, AnyType Expr2)` This function is the same as `strcmp`, except that letter case is *not* significant.

`String toUpper(AnyType Expr)` The single argument is converted to type `String` by function `string(Expr)`. The return value is this string, with all lower case letters converted to upper case. If the argument evaluates to `ERROR` or `UNDEFINED`, `ERROR` is returned.

This function returns `ERROR` if greater than 1 argument is given.

`String toLower(AnyType Expr)` The single argument is converted to type `String` by function `string(Expr)`. The return value is this string, with all upper case letters converted to lower case. If the argument evaluates to `ERROR` or `UNDEFINED`, `ERROR` is returned.

This function returns `ERROR` if other than exactly 1 argument is given.

`Integer size(AnyType Expr)` Returns the number of characters in the string, after calling function `string(Expr)`. If the argument evaluates to `ERROR` or `UNDEFINED`, `ERROR` is returned.

This function returns `ERROR` if other than exactly 1 argument is given.

`Integer time()` Returns the current coordinated universal time, which is the same as the ClassAd attribute `CurrentTime`. This is the time, in seconds, since midnight of January 1, 1970.

`String formatTime([ Integer time ] [ , String format ])` Returns a formatted string that is a representation of `time`. The argument `time` is interpreted as coordinated universe time in seconds, since midnight of January 1, 1970. If not specified, `time` will default to the value of attribute `CurrentTime`.

The argument `format` is interpreted similarly to the format argument of the ANSI C strftime function. It consists of arbitrary text plus placeholders for elements of the time. These placeholders are percent signs (%) followed by a single letter. To have a percent sign in the output, use a double percent sign (%%). If `format` is not specified, it defaults to `%c`.

Because the implementation uses `strftime()` to implement this, and some versions implement extra, non-ANSI C options, the exact options available to an implementation may vary. An implementation is only required to implement the ANSI C options, which are:

`%a` abbreviated weekday name

`%A` full weekday name

`%b` abbreviated month name

`%B` full month name

`%c` local date and time representation

`%d` day of the month (01-31)

`%H` hour in the 24-hour clock (0-23)

`%I` hour in the 12-hour clock (01-12)

`%j` day of the year (001-366)

`%m` month (01-12)

`%M` minute (00-59)

`%p` local equivalent of AM or PM

`%S` second (00-59)

`%U` week number of the year (Sunday as first day of week) (00-53)

`%w` weekday (0-6, Sunday is 0)

`%W` week number of the year (Monday as first day of week) (00-53)

`%x` local date representation

`%X` local time representation

`%y` year without century (00-99)

`%Y` year with century

`%Z` time zone name, if any

`String interval(Integer seconds)` Uses `seconds` to return a string of the form `days+hh:mm:ss`. This represents an interval of time. Leading values that are zero are omitted from the string. For example, `seconds` of 67 becomes "1:07". A second example, `seconds` of 1472523 = 17*24*60*60 + 1*60*60 + 2*60 + 3, results in the string "17+1:02:03".

`AnyType debug(AnyType expression)` This function evaluates its argument, and it returns the result. Thus, it is a no-operation. However, a side-effect of the function is that information about the evaluation is logged to the evaluating program's log file. This is useful for determining why a given ClassAd expression is evaluating the way it does. For example, if a *condor_startd* `START` expression is unexpectedly evaluating to `UNDEFINED`, then wrapping the expression in this `debug()` function will log information about each component of the expression to the log file, making it easier to understand the expression.

For the following functions, a delimiter is represented by a string. Each character within the delimiter string delimits individual strings within a list of strings that is given by a single string. The default delimiter contains the comma and space characters. A string within the list is ended (delimited) by one or more characters within the delimiter string.

Integer stringListSize(String list [, String delimiter]) Returns the number of elements in the string `list`, as delimited by the optional `delimiter` string. Returns `ERROR` if either argument is not a string.

This function returns `ERROR` if other than 1 or 2 arguments are given.

Integer stringListSum(String list [, String delimiter])

OR Real stringListSum(String list [, String delimiter]) Sums and returns the sum of all items in the string `list`, as delimited by the optional `delimiter` string. If all items in the list are integers, the return value is also an integer. If any item in the list is a real value (noninteger), the return value is a real. If any item does not represent an integer or real value, the return value is `ERROR`.

Real stringListAvg(String list [, String delimiter]) Sums and returns the real-valued average of all items in the string `list`, as delimited by the optional `delimiter` string. If any item does not represent an integer or real value, the return value is `ERROR`. A list with 0 items (the empty list) returns the value 0.0.

Integer stringListMin(String list [, String delimiter])

OR Real stringListMin(String list [, String delimiter]) Finds and returns the minimum value from all items in the string `list`, as delimited by the optional `delimiter` string. If all items in the list are integers, the return value is also an integer. If any item in the list is a real value (noninteger), the return value is a real. If any item does not represent an integer or real value, the return value is `ERROR`. A list with 0 items (the empty list) returns the value `UNDEFINED`.

Integer stringListMax(String list [, String delimiter])

OR Real stringListMax(String list [, String delimiter]) Finds and returns the maximum value from all items in the string `list`, as delimited by the optional `delimiter` string. If all items in the list are integers, the return value is also an integer. If any item in the list is a real value (noninteger), the return value is a real. If any item does not represent an integer or real value, the return value is `ERROR`. A list with 0 items (the empty list) returns the value `UNDEFINED`.

Boolean stringListMember(String x, String list [, String delimiter]) Returns `TRUE` if item `x` is in the string `list`, as delimited by the optional `delimiter` string. Returns `FALSE` if item `x` is not in the string `list`. Comparison is done with `strcmp()`. The return value is `ERROR`, if any of the arguments are not strings.

Boolean stringListIMember(String x, String list [, String delimiter]) Same as `stringListMember()`, but comparison is done with `stricmp()`, so letter case is not relevant.

Boolean stringList_regexpMember(String pattern, String list [, String delimiter])
Returns TRUE if the string `pattern` is a regular expression that matches an item in the string `list`, as delimited by the optional `delimiter` string. String `options` modifies how the match is performed. Returns FALSE if `pattern` does not match any entries in `list`. The return value is ERROR, if any of the arguments are not strings, or if `pattern` is not a valid regular expression.

The following three functions utilize regular expressions as defined and supported by the PCRE library. See http://www.pcre.org for complete documentation of regular expressions.

The `options` argument to these functions is a string of special characters that modify the use of the regular expressions. Inclusion of characters other than these as options are ignored.

I or i Ignore letter case.

M or m Modifies the interpretation of the carat (^) and dollar sign ($) characters. The carat character matches the start of a string, as well as after each newline character. The dollar sign character matches before a newline character.

S or s The period matches any character, including the newline character.

Boolean regexp(String pattern, String target [, String options])
Returns TRUE if the string `target` is a regular expression as described by `pattern`. Returns FALSE otherwise. If any argument is not a string, or if `pattern` does not describe a valid regular expression, returns ERROR.

String regexps(String pattern, String target, String substitute,

[String options]) The regular expression `pattern` is applied to `target`. If the string `target` is a regular expression as described by `pattern`, the string `substitute` is returned, with backslash expansion performed. The return value is ERROR, if any of the arguments are not strings.

Boolean stringListRegexpMember(String pattern, String list [, String delimiter]

[, String options]) Returns TRUE if any of the strings within the `list` is a regular expression as described by `pattern`. Returns FALSE otherwise. If any argument is not a string, or if `pattern` does not describe a valid regular expression, returns ERROR. To include the fourth (optional) argument `options`, a third argument of `delimiter` is required. A default value for a delimiter is " ,".

4.1.3 Old ClassAd Evaluation Semantics

The ClassAd mechanism's primary purpose is for matching entities that supply constraints on candidate matches. The mechanism is therefore defined to carry out expression evaluations in the

context of two ClassAds that are testing each other for a potential match. For example, the *condor_negotiator* evaluates the `Requirements` expressions of machine and job ClassAds to test if they can be matched. The semantics of evaluating such constraints is defined below.

Literals

Literals are self-evaluating, Thus, integer, string, real, undefined and error values evaluate to themselves.

Attribute References

Since the expression evaluation is being carried out in the context of two ClassAds, there is a potential for name space ambiguities. The following rules define the semantics of attribute references made by ad A that is being evaluated in a context with another ad B:

1. If the reference is prefixed by a scope resolution prefix,
   - If the prefix is `MY.`, the attribute is looked up in ClassAd A. If the named attribute does not exist in A, the value of the reference is `UNDEFINED`. Otherwise, the value of the reference is the value of the expression bound to the attribute name.
   - Similarly, if the prefix is `TARGET.`, the attribute is looked up in ClassAd B. If the named attribute does not exist in B, the value of the reference is `UNDEFINED`. Otherwise, the value of the reference is the value of the expression bound to the attribute name.
2. If the reference is not prefixed by a scope resolution prefix,
   - If the attribute is defined in A, the value of the reference is the value of the expression bound to the attribute name in A.
   - Otherwise, if the attribute is defined in B, the value of the reference is the value of the expression bound to the attribute name in B.
   - Otherwise, if the attribute is defined in the ClassAd environment, the value from the environment is returned. This is a special environment, to be distinguished from the Unix environment. Currently, the only attribute of the environment is `CurrentTime`, which evaluates to the integer value returned by the system call `time(2)`.
   - Otherwise, the value of the reference is `UNDEFINED`.
3. Finally, if the reference refers to an expression that is itself in the process of being evaluated, there is a circular dependency in the evaluation. The value of the reference is `ERROR`.

Operators

All operators in the ClassAd language are *total*, and thus have well defined behavior regardless of the supplied operands. Furthermore, most operators are *strict* with respect to `ERROR` and `UNDEFINED`, and thus evaluate to `ERROR` or `UNDEFINED` if either of their operands have these exceptional values.

- **Arithmetic operators:**
  1. The operators `*`, `/`, `+` and `-` operate arithmetically only on integers and reals.
  2. Arithmetic is carried out in the same type as both operands, and type promotions from integers to reals are performed if one operand is an integer and the other real.
  3. The operators are strict with respect to both `UNDEFINED` and `ERROR`.
  4. If either operand is not a numerical type, the value of the operation is `ERROR`.
- **Comparison operators:**
  1. The comparison operators `==`, `!=`, `<=`, `<`, `>=` and `>` operate on integers, reals and strings.
  2. String comparisons are case insensitive for most operators. The only exceptions are the operators `=?=` and `=!=`, which do case sensitive comparisons assuming both sides are strings.
  3. Comparisons are carried out in the same type as both operands, and type promotions from integers to reals are performed if one operand is a real, and the other an integer. Strings may not be converted to any other type, so comparing a string and an integer or a string and a real results in `ERROR`.
  4. The operators `==`, `!=`, `<=`, `<` and `>=` `>` are strict with respect to both `UNDEFINED` and `ERROR`.
  5. In addition, the operators `=?=` and `=!=` behave similar to `==` and `!=`, but are not strict. Semantically, the `=?=` tests if its operands are "identical," i.e., have the same type and the same value. For example, `10 == UNDEFINED` and `UNDEFINED == UNDEFINED` both evaluate to `UNDEFINED`, but `10 =?= UNDEFINED` and `UNDEFINED =?= UNDEFINED` evaluate to `FALSE` and `TRUE` respectively. The `=!=` operator test for the "is not identical to" condition.
- **Logical operators:**
  1. The logical operators `&&` and `||` operate on integers and reals. The zero value of these types are considered `FALSE` and non-zero values `TRUE`.
  2. The operators are *not* strict, and exploit the "don't care" properties of the operators to squash `UNDEFINED` and `ERROR` values when possible. For example, `UNDEFINED && FALSE` evaluates to `FALSE`, but `UNDEFINED || FALSE` evaluates to `UNDEFINED`.
  3. Any string operand is equivalent to an `ERROR` operand for a logical operator. In other words, `TRUE && "foobar"` evaluates to `ERROR`.

Expression Examples

The `=?=` operator is similar to the `==` operator. It checks if the left hand side operand is identical in both type and value to the the right hand side operand, returning `TRUE` when they are identical. A key point in understanding is that the `=?=` operator only produces evaluation results of `TRUE` and `FALSE`, where the `==` operator may produce evaluation results `TRUE`, `FALSE`, `UNDEFINED`, or

ERROR. Table 4.1 presents examples that define the outcome of the == operator. Table 4.2 presents examples that define the outcome of the =?= operator.

expression	evaluated result
(10 == 10)	TRUE
(10 == 5)	FALSE
(10 == "ABC")	ERROR
(10 == UNDEFINED)	UNDEFINED
(UNDEFINED == UNDEFINED)	UNDEFINED

Table 4.1: Evaluation examples for the == operator

expression	evaluated result
(10 =?= 10)	TRUE
(10 =?= 5)	FALSE
(10 =?= "ABC")	FALSE
(10 =?= UNDEFINED)	FALSE
(UNDEFINED =?= UNDEFINED)	TRUE

Table 4.2: Evaluation examples for the =?= operator

The =!= operator is similar to the != operator. It checks if the left hand side operand is *not* identical in both type and value to the the right hand side operand, returning FALSE when they are identical. A key point in understanding is that the =!= operator only produces evaluation results of TRUE and FALSE, where the != operator may produce evaluation results TRUE, FALSE, UNDEFINED, or ERROR. Table 4.3 presents examples that define the outcome of the != operator. Table 4.4 presents examples that define the outcome of the =!= operator.

expression	evaluated result
(10 != 10)	FALSE
(10 != 5)	TRUE
(10 != "ABC")	ERROR
(10 != UNDEFINED)	UNDEFINED
(UNDEFINED != UNDEFINED)	UNDEFINED

Table 4.3: Evaluation examples for the != operator

expression	evaluated result
(10 =!= 10)	FALSE
(10 =!= 5)	TRUE
(10 =!= "ABC")	TRUE
(10 =!= UNDEFINED)	TRUE
(UNDEFINED =!= UNDEFINED)	FALSE

Table 4.4: Evaluation examples for the =!= operator

4.1.4 Old ClassAds in the Condor System

The simplicity and flexibility of ClassAds is heavily exploited in the Condor system. ClassAds are not only used to represent machines and jobs in the Condor pool, but also other entities that exist in the pool such as checkpoint servers, submitters of jobs and master daemons. Since arbitrary expressions may be supplied and evaluated over these ads, users have a uniform and powerful mechanism to specify constraints over these ads. These constraints can take the form of `Requirements` expressions in resource and job ads, or queries over other ads.

Constraints and Preferences

The `requirements` and `rank` expressions within the submit description file are the mechanism by which users specify the constraints and preferences of jobs. For machines, the configuration determines both constraints and preferences of the machines.

For both machine and job, the `rank` expression specifies the desirability of the match (where higher numbers mean better matches). For example, a job ad may contain the following expressions:

```
Requirements = Arch=="INTEL" && OpSys == "LINUX"
Rank         = TARGET.Memory + TARGET.Mips
```

In this case, the job requires an Intel 32-bit computer running RHEL 3 as its operating system. Among all such computers, the customer prefers those with large physical memories and high MIPS ratings. Since the `Rank` is a user-specified metric, *any* expression may be used to specify the perceived desirability of the match. The *condor_negotiator* daemon runs algorithms to deliver the best resource (as defined by the `rank` expression) while satisfying other required criteria.

Similarly, the machine may place constraints and preferences on the jobs that it will run by setting the machine's configuration. For example,

```
Friend        = Owner == "tannenba" || Owner == "wright"
ResearchGroup = Owner == "jbasney" || Owner == "raman"
Trusted       = Owner != "rival" && Owner != "riffraff"
START         = Trusted && ( ResearchGroup || LoadAvg < 0.3 &&
                      KeyboardIdle > 15*60 )
RANK          = Friend + ResearchGroup*10
```

The above policy states that the computer will never run jobs owned by users rival and riffraff,

while the computer will always run a job submitted by members of the research group. Furthermore, jobs submitted by friends are preferred to other foreign jobs, and jobs submitted by the research group are preferred to jobs submitted by friends.

Note: Because of the dynamic nature of ClassAd expressions, there is no *a priori* notion of an integer-valued expression, a real-valued expression, etc. However, it is intuitive to think of the `Requirements` and `Rank` expressions as integer-valued and real-valued expressions, respectively. If the actual type of the expression is not of the expected type, the value is assumed to be zero.

Querying with ClassAd Expressions

The flexibility of this system may also be used when querying ClassAds through the *condor_status* and *condor_q* tools which allow users to supply ClassAd constraint expressions from the command line.

Needed syntax is different on Unix and Windows platforms, due to the interpretation of characters in forming command-line arguments. The expression must be a single command-line argument, and the resulting examples differ for the platforms. For Unix shells, single quote marks are used to delimit a single argument. For a Windows command window, double quote marks are used to delimit a single argument. Within the argument, Unix escapes the double quote mark by prepending a backslash to the double quote mark. Windows escapes the double quote mark by prepending another double quote mark. There may not be spaces in between.

Here are several examples. To find all computers which have had their keyboards idle for more than 20 minutes and have more than 100 MB of memory, the desired ClassAd expression is

```
KeyboardIdle > 20*60 && Memory > 100
```

On a Unix platform, the command appears as

```
% condor_status -const 'KeyboardIdle > 20*60 && Memory > 100'

Name       Arch     OpSys        State      Activity   LoadAv Mem  ActvtyTime

amul.cs.wi SUN4u    SOLARIS251   Claimed    Busy       1.000  128   0+03:45:01
aura.cs.wi SUN4u    SOLARIS251   Claimed    Busy       1.000  128   0+00:15:01
balder.cs. INTEL    SOLARIS251   Claimed    Busy       1.000  1024  0+01:05:00
beatrice.c INTEL    SOLARIS251   Claimed    Busy       1.000  128   0+01:30:02
...
...
                     Machines Owner Claimed Unclaimed Matched Preempting

    SUN4u/SOLARIS251        3     0       3         0       0          0
    INTEL/SOLARIS251       21     0      21         0       0          0
    SUN4x/SOLARIS251        3     0       3         0       0          0
       INTEL/WINNT51        1     0       0         1       0          0
         INTEL/LINUX        1     0       1         0       0          0

               Total       29     0      28         1       0          0
```

The Windows equivalent command is

```
>condor_status -const "KeyboardIdle > 20*60 && Memory > 100"
```

Here is an example for a Unix platform that utilizes a regular expression ClassAd function to list specific information. A file contains ClassAd information. *condor_advertise* is used to inject this information, and *condor_status* constrains the search with an expression that contains a ClassAd function.

```
% cat ad
MyType = "Generic"
FauxType = "DBMS"
Name = "random-test"
Machine = "f05.cs.wisc.edu"
MyAddress = "<128.105.149.105:34000>"
DaemonStartTime = 1153192799
UpdateSequenceNumber = 1

% condor_advertise UPDATE_AD_GENERIC ad

% condor_status -any -constraint 'FauxType=="DBMS" && regexp("random.*", Name, "i")'

MyType               TargetType           Name

Generic              None                 random-test
```

The ClassAd expression describing a machine that advertises a Windows NT operating system:

```
OpSys == "WINNT51"
```

Here are three equivalent ways on a Unix platform to list all machines advertising a Windows NT operating system. Spaces appear in these examples to show where they are permitted.

```
% condor_status -constraint ' OpSys == "WINNT51"  '


% condor_status -constraint OpSys==\"WINNT51\"


% condor_status -constraint "OpSys==\"WINNT51\""
```

The equivalent command on a Windows platform to list all machines advertising a Windows NT operating system must delimit the single argument with double quote marks, and then escape the needed double quote marks that identify the string within the expression. Spaces appear in this example where they are permitted.

```
>condor_status -constraint " OpSys == ""WINNT51"" "
```

4.2 Condor's Checkpoint Mechanism

Checkpointing is taking a snapshot of the current state of a program in such a way that the program can be restarted from that state at a later time. Checkpointing gives the Condor scheduler the freedom to reconsider scheduling decisions through preemptive-resume scheduling. If the scheduler decides to no longer allocate a machine to a job (for example, when the owner of that machine returns), it can checkpoint the job and preempt it without losing the work the job has already accomplished. The job can be resumed later when the scheduler allocates it a new machine. Additionally, periodic checkpointing provides fault tolerance in Condor. Snapshots are taken periodically, and after an interruption in service the program can continue from the most recent snapshot.

Condor provides checkpointing services to single process jobs on a number of Unix platforms. To enable checkpointing, the user must link the program with the Condor system call library (`libcondorsyscall.a`), using the *condor_compile* command. This means that the user must have the object files or source code of the program to use Condor checkpointing. However, the checkpointing services provided by Condor are strictly optional. So, while there are some classes of jobs for which Condor does not provide checkpointing services, these jobs may still be submitted to Condor to take advantage of Condor's resource management functionality. (See section 2.4.1 on page 15 for a description of the classes of jobs for which Condor does not provide checkpointing services.)

Process checkpointing is implemented in the Condor system call library as a signal handler. When Condor sends a checkpoint signal to a process linked with this library, the provided signal handler writes the state of the process out to a file or a network socket. This state includes the contents of the process stack and data segments, all shared library code and data mapped into the process's address space, the state of all open files, and any signal handlers and pending signals. On restart, the process reads this state from the file, restoring the stack, shared library and data segments, file state, signal handlers, and pending signals. The checkpoint signal handler then returns to user code, which continues from where it left off when the checkpoint signal arrived.

Condor processes for which checkpointing is enabled perform a checkpoint when preempted from a machine. When a suitable replacement execution machine is found (of the same architecture and operating system), the process is restored on this new machine from the checkpoint, and computation is resumed from where it left off. Jobs that can not be checkpointed are preempted and restarted from the beginning.

Condor's periodic checkpointing provides fault tolerance. Condor pools are each configured with the `PERIODIC_CHECKPOINT` expression which controls when and how often jobs which can be checkpointed do periodic checkpoints (examples: never, every three hours, etc.). When the time for a periodic checkpoint occurs, the job suspends processing, performs the checkpoint, and immediately continues from where it left off. There is also a *condor_ckpt* command which allows the user to request that a Condor job immediately perform a periodic checkpoint.

In all cases, Condor jobs continue execution from the most recent complete checkpoint. If service is interrupted while a checkpoint is being performed, causing that checkpoint to fail, the process will restart from the previous checkpoint. Condor uses a commit style algorithm for writing checkpoints: a previous checkpoint is deleted only after a new complete checkpoint has been written

successfully.

In certain cases, checkpointing may be delayed until a more appropriate time. For example, a Condor job will defer a checkpoint request if it is communicating with another process over the network. When the network connection is closed, the checkpoint will occur.

The Condor checkpointing facility can also be used for any Unix process outside of the Condor batch environment. Standalone checkpointing is described in section 4.2.1.

Condor can produce and use compressed checkpoints. Configuration variables (detailed in section 3.3.12 control whether compression is used. The default is to not compress.

By default, a checkpoint is written to a file on the local disk of the machine where the job was submitted. A Condor pool can also be configured with a checkpoint server or servers that serve as a repository for checkpoints. (See section 3.8 on page 384.) When a host is configured to use a checkpoint server, jobs submitted on that machine write and read checkpoints to and from the server rather than the local disk of the submitting machine, taking the burden of storing checkpoint files off of the submitting machines and placing it instead on server machines (with disk space dedicated to the purpose of storing checkpoints).

4.2.1 Standalone Checkpointing

Using the Condor checkpoint library without the remote system call functionality and outside of the Condor system is known as standalone mode checkpointing.

To prepare a program for standalone checkpointing, simply use the *condor_compile* utility as for a standard Condor job, but do not use *condor_submit*. Run the program from the command line. The checkpointing library will print a message to let you know that checkpointing is enabled and to inform you of the default name for the checkpoint image. The message is of the form:

```
Condor: Notice: Will checkpoint to program_name.ckpt
Condor: Notice: Remote system calls disabled.
```

Platforms that use address space randomization will need a modified invocation of the program, as described in section 6.1.2 on page 594. The invocation disables the address space randomization.

To force the program to write a checkpoint image and stop, send it the SIGTSTP signal or press control-Z. To force the program to write a checkpoint image and continue executing, send it the SIGUSR2 signal.

To restart a program using a checkpoint, run the program with the argument *-_condor_restart* followed by the name of the checkpoint image file. As an example, if the program is called *P1* and the checkpoint is called `P1.ckpt`, use

```
P1 -_condor_restart P1.ckpt
```

Again, platforms that implement address space randomization will need a modified invocation, as described in section 6.1.2.

4.2.2 Checkpoint Safety

Some programs have fundamental limitations that make them unsafe for checkpointing. For example, a program that both reads and writes a single file may enter an unexpected state. Here is an example of how this might happen.

1. Record a checkpoint image.
2. Read data from a file.
3. Write data to the same file.
4. Execution failure, so roll back to step 2.

In this example, the program would re-read data from the file, but instead of finding the original data, would see data created in the future, and yield unexpected results.

To prevent this sort of accident, Condor displays a warning if a file is used for both reading and writing. You can ignore or disable these warnings if you choose (see section 4.2.3,) but please understand that your program may compute incorrect results.

4.2.3 Checkpoint Warnings

Condor has warning messages in the case unexpected behaviors in your program. For example, if file `x` is opened for reading and writing, you will see:

```
Condor: Warning: READWRITE: File '/tmp/x' used for both reading and writing.
```

You may control how these messages are displayed with the `-_condor_warning` command-line argument. This argument accepts a warning category and a mode. The category describes a certain class of messages, such as READWRITE or ALL. The mode describes what to do with the category. It may be ON, OFF, or ONCE. If a category is ON, it is always displayed. If a category is OFF, it is never displayed. If a category is ONCE, it is displayed only once. To show all the available categories and modes, just use `-_condor_warning` with no arguments.

For example, to limit read/write warnings to one instance:

```
-_condor_warning READWRITE ONCE
```

To turn all ordinary notices off:

```
-_condor_warning NOTICE OFF
```

The same effect can be accomplished within a program by using the function `_condor_warning_config`, described in section 4.2.4.

4.2.4 Checkpoint Library Interface

A program need not be rewritten to take advantage of checkpointing. However, the checkpointing library provides several C entry points that allow for a program to control its own checkpointing behavior if needed.

- `void init_image_with_file_name( char *ckpt_file_name )`
  This function explicitly sets a file name to use when producing or using a checkpoint. `ckpt()` or `ckpt_and_exit()` must be called to produce the checkpoint, and `restart()` must be called to perform the actual restart.
- `void init_image_with_file_descriptor( int fd )`
  This function explicitly sets a file descriptor to use when producing or using a checkpoint. `ckpt()` or `ckpt_and_exit()` must be called to produce the checkpoint, and `restart()` must be called to perform the actual restart.
- `void ckpt()`
  This function causes a checkpoint image to be written to disk. The program will continue to execute. This is identical to sending the program a SIGUSR2 signal.
- `void ckpt_and_exit()`
  This function causes a checkpoint image to be written to disk. The program will then exit. This is identical to sending the program a SIGTSTP signal.
- `void restart()`
  This function causes the program to read the checkpoint image and to resume execution of the program from the point where the checkpoint was taken. This function does not return.
- `void _condor_ckpt_disable()`
  This function temporarily disables checkpointing. This can be handy if your program does something that is not checkpoint-safe. For example, if a program must not be interrupted while accessing a special file, call `_condor_ckpt_disable()`, access the file, and then call `_condor_ckpt_enable()`. Some program actions, such as opening a socket or a pipe, implicitly cause checkpointing to be disabled.
- `void _condor_ckpt_enable()`
  This function re-enables checkpointing after a call to `_condor_ckpt_disable()`. If a checkpointing signal arrived while checkpointing was disabled, the checkpoint will occur when this function is called. Disabling and enabling of checkpointing must occur in matched pairs. `_condor_ckpt_enable()` must be called once for every time that `_condor_ckpt_disable()` is called.

- `int _condor_warning_config( const char *kind, const char *mode )`
  This function controls what warnings are displayed by Condor. The `kind` and `mode` arguments are the same as for the `-_condor_warning` option described in section 4.2.3. This function returns true if the arguments are understood and accepted. Otherwise, it returns false.
- `extern int condor_compress_ckpt`
  Setting this variable to one causes checkpoint images to be compressed. Setting it to zero disables compression.

4.3 Computing On Demand (COD)

Computing On Demand (COD) extends Condor's high throughput computing abilities to include a method for running short-term jobs on instantly-available resources.

The motivation for COD extends Condor's job management to include interactive, compute-intensive jobs, giving these jobs immediate access to the compute power they need over a relatively short period of time. COD provides computing power *on demand*, switching predefined resources from working on Condor jobs to working on the COD jobs. These COD jobs (applications) cannot use the batch scheduling functionality of Condor, since the COD jobs require interactive response-time. Many of the applications that are well-suited to Condor's COD capabilities involve a cycle: application blocked on user input, computation burst to compute results, block again on user input, computation burst, etc. When the resources are not being used for the bursts of computation to service the application, they should continue to execute long-running batch jobs.

Here are examples of applications that may benefit from COD capability:

- A giant spreadsheet with a large number of highly complex formulas which take a lot of compute power to recalculate. The spreadsheet application (as a COD application) predefines a claim on resources within the Condor pool. When the user presses a `recalculate` button, the predefined Condor resources (nodes) work on the computation and send the results back to the master application providing the user interface and displaying the data. Ideally, while the user is entering new data or modifying formulas, these nodes work on non-COD jobs.
- A graphics rendering application that waits for user input to select an image to render. The rendering requires a huge burst of computation to produce the image. Examples are various Computer-Aided Design (CAD) tools, fractal rendering programs, and ray-tracing tools.
- Visualization tools for data mining.

The way Condor helps these kinds of applications is to provide an infrastructure to use Condor batch resources for the types of compute nodes described above. Condor does *NOT* provide tools to parallelize existing GUI applications. The COD functionality is an interface to allow these compute nodes to interact with long-running Condor batch jobs. The user provides both the compute

node applications and the interactive master application that controls them. Condor only provides a mechanism to allow these interactive (and often parallelized) applications to seamlessly interact with the Condor batch system.

4.3.1 Overview of How COD Works

The resources of a Condor pool (nodes) run jobs. When a high-priority COD job appears at a node, the lower-priority (currently running) batch job is suspended. The COD job runs immediately, while the batch job remains suspended. When the COD job completes, the batch job instantly resumes execution.

Administratively, an interactive COD application puts claims on nodes. While the COD application does not need the nodes (to run the COD jobs), the claims are suspended, allowing batch jobs to run.

4.3.2 Authorizing Users to Create and Manage COD Claims

Claims on nodes are assigned to users. A user with a claim on a resource can then suspend and resume a COD job at will. This gives the user a great deal of power on the claimed resource, even if it is owned by another user. Because of this, it is essential that users allowed to claim COD resources can be trusted not to abuse this power. Users are authorized to have access to the privilege of creating and using a COD claim on a machine. This privilege is granted when the Condor administrator places a given user name in the `VALID_COD_USERS` list in the Condor configuration for the machine (usually in a local configuration file).

In addition, the tools to request and manage COD claims require that the user issuing the commands be authenticated. Use one of the strong authentication methods described in section 3.6.1 “Security Configuration” on page 315. If one of these methods cannot be used, then file system authentication may be used when directly logging in to that machine (to be claimed) and issuing the command locally.

4.3.3 Defining a COD Application

To run an application on a claimed COD resource, an authorized user defines characteristics of the application. Examples of characteristics are the executable or script to use, the directory to run the application in, command-line arguments, and files to use for standard input and output. COD users specify a ClassAd that describes these characteristics for their application. There are two ways for a user to define a COD application’s ClassAd:

1. in the Condor configuration files of the COD resources
2. when they use the *condor_cod* command-line tool to launch the application itself

These two methods for defining the ClassAd can be used together. For example, the user can define some attributes in the configuration file, and only provide a few dynamically defined attributes with the *condor_cod* tool.

Regardless of how the COD application's ClassAd is defined, the application's executable and input data must be pre-staged at the node. This is a current limitation of Condor's support for COD that will eventually go away. For now, there is no mechanism to transfer files for a COD application, and all I/O must be performed locally or onto a network file system that is accessible by a node.

The following three sections detail defining the attributes. The first lists the attributes that can be used to define a COD application. The second describes how to define these attributes in a Condor configuration file. The third explains how to define these attributes using the *condor_cod* tool.

COD Application Attributes

Attributes for a COD application are either required or optional. The following attributes are *required*:

Cmd This attribute defines the full path to the executable program to be run as a COD application. Since Condor does not currently provide any mechanism to transfer files on behalf of COD applications, this path should be a valid path on the machine where the application will be run. It is a string attribute, and must therefore be enclosed in quotation marks (`"`). There is no default.

JobUniverse This attribute defines what Condor job universe to use for the given COD application. At this point, the only supported universes are vanilla and Java. This attribute must be an integer, with vanilla using the value 5, and Java the value 10. This attribute is required if the default `START` expression is used, as it references the attribute `IsValidCheckpointPlatform`, which needs the job universe.

Owner If the *condor_startd* daemon is executing as root on the resource where a COD application will run, the user must also define `Owner` to specify what user name the application will run as. (On Windows, the *condor_startd* daemon always runs as an Administrator service, which is equivalent to running as root on UNIX platforms). If the user specifies any COD application attributes with the *condor_cod_activate* command-line tool, the `Owner` attribute will be defined as the user name that ran *condor_cod_activate*. However, if the user defines all attributes of their COD application in the Condor configuration files, and does not define any attributes with the *condor_cod_activate* command-line tool (both methods are described below in more detail), there is no default and `Owner` must be specified in the configuration file. `Owner` must contain a valid user name on the given COD resource. It is a string attribute, and must therefore be enclosed in quotation marks (`"`).

The following list of attributes are *optional*:

IWD IWD is an acronym for Initial Working Directory. It defines the full path to the directory where a given COD application are to be run. Unless the application changes its current working

directory, any relative path names used by the application will be relative to the IWD. If any other attributes that define file names (for example, `In`, `Out`, and so on) do not contain a full path, the `IWD` will automatically be pre-pended to those file names. It is a string attribute, and must therefore be enclosed in quotation marks (`"`). If the `IWD` is not specified, the temporary execution sandbox created by the *condor_starter* will be used as the initial working directory.

`In` This string defines the path to the file on the COD resource that should be used as standard input (`stdin`) for the COD application. This file (and all parent directories) must be readable by whatever user the COD application will run as. If not specified, the default is `/dev/null`.

`Out` This string defines the path to the file on the COD resource that should be used as standard output (`stdout`) for the COD application. This file must be writable (and all parent directories readable) by whatever user the COD application will run as. If not specified, the default is `/dev/null`. It is a string attribute, and must therefore be enclosed in quotation marks (`"`).

`Err` This string defines the path to the file on the COD resource that should be used as standard error (`stderr`) for the COD application. This file must be writable (and all parent directories readable) by whatever user the COD application will run as. If not specified, the default is `/dev/null`. It is a string attribute, and must therefore be enclosed in quotation marks (`"`).

`Env` This string defines environment variables to set for a given COD application. Each environment variable has the form `NAME=value`. Multiple variables are delimited with a semicolon. An example: `Env = "PATH=/usr/local/bin:/usr/bin;TERM=vt100"` It is a string attribute, and must therefore be enclosed in quotation marks (`"`).

`Args` This string attribute defines the list of arguments to be supplied to the program on the command-line. The arguments are delimited (separated) by space characters. There is no default. If the `JobUniverse` corresponds to the Java universe, the first argument must be the name of the class containing `main`. It is a string attribute, and must therefore be enclosed in quotation marks (`"`).

`JarFiles` This string attribute is only used if `JobUniverse` is 10 (the Java universe). If a given COD application is a Java program, specify the JAR files that the program requires with this attribute. There is no default. It is a string attribute, and must therefore be enclosed in quotation marks (`"`). Multiple file names may be delimited with either commas or white space characters, and therefore, file names can not contain spaces.

`KillSig` This attribute specifies what signal should be sent whenever the Condor system needs to gracefully shutdown the COD application. It can either be specified as a string containing the signal name (for example `KillSig = "SIGQUIT"`), or as an integer (`KillSig = 3`) The default is to use SIGTERM.

`StarterUserLog` This string specifies a file name for a log file that the *condor_starter* daemon can write with entries for relevant events in the life of a given COD application. It is similar to the UserLog file specified for regular Condor jobs with the `Log` setting in a submit description file. However, certain attributes that are placed in the regular UserLog file do not make sense in the COD environment, and are therefore omitted. The default is not to write this log file. It is a string attribute, and must therefore be enclosed in quotation marks (`"`).

StarterUserLogUseXML If the StarterUserLog attribute is defined, the default format is a human-readable format. However, Condor can write out this log in an XML representation, instead. To enable the XML format for this UserLog, the StarterUserLogUseXML boolean is set to TRUE. The default if not specified is FALSE.

NOTE: If any path attribute (Cmd, In, Out,Err, StarterUserLog) is not a full path name, Condor automatically prepends the value of IWD.

The final set of attributes define an identification for a COD application. The job ID is made up of both the ClusterId and ProcId attributes (as described below). This job ID is similar to the job ID that is created whenever a regular Condor batch job is submitted. For regular Condor batch jobs, the job ID is assigned automatically by the *condor_schedd* whenever a new job is submitted into the persistent job queue. However, since there is no persistent job queue for COD, the usual mechanism to identify the jobs does not exist. Moreover, commands that require the job ID for batch jobs such as *condor_q* and *condor_rm* do not exist for COD. Instead, the claim ID is the unique identifier for COD jobs and COD-related commands.

When using COD, the job ID is only used to identify the job in various log messages and in the COD-specific output of *condor_status*. The COD job ID is part of the information included in all events written to the StarterUserLog regarding a given job. The COD job ID is also used in the Condor debugging logs described in section 3.3.4 on page 170 For example, in the *condor_starter* daemon's log file for COD jobs (called StarterLog.cod by default) or in the *condor_startd* daemon's log file (called StartLog by default).

These COD IDs are optional. The job ID is useful to define where it helps a user with accounting or debugging of their own application. In this case, it is the user's responsibility to ensure uniqueness, if so desired.

ClusterId This integer defines the cluster identifier for a COD job. The default value is 1. The ClusterId can also be defined with the *condor_cod_activate* command-line tool using the **-cluster** option.

ProcId This integer defines the process identifier (within a cluster) for a COD job. The default value is 0. The ProcId can also be defined with the *condor_cod_activate* command-line tool using the **-cluster** option.

NOTE: The cluster and proc identifiers can also be specified as command-line arguments to the *condor_cod_activate* tool when spawning a given COD application. See section 4.3.4 below for details on using *condor_cod_activate*.

Defining Attributes in the Condor Configuration Files

To define COD attributes in the Condor configuration file for a given application, the user selects a keyword to uniquely name ClassAd attributes of the application. This case-insensitive keyword is used as a prefix for the various configuration file attribute names. When a user wishes to spawn a

given application, the keyword is given as an argument to the *condor_cod* tool and the keyword is used at the remote COD resource to find attributes which define the application.

Any of the ClassAd attributes described in the previous section can be specified in the configuration file with the keyword prefix followed by an underscore character (`"_"`).

For example, if the user's keyword for a given fractal generation application is "FractGen", the resulting entries in the Condor configuration file may appear as:

```
FractGen_Cmd = "/usr/local/bin/fractgen"
FractGen_Iwd = "/tmp/cod-fractgen"
FractGen_Out = "/tmp/cod-fractgen/output"
FractGen_Err = "/tmp/cod-fractgen/error"
FractGen_Args = "mandelbrot -0.65865,-0.56254 -0.45865,-0.71254"
```

In this example, the executable may create other files. The `Out` and `Err` attributes specified in the configuration file are only for standard output and standard error redirection.

When the user wishes to spawn an instance of this application, they use the **-keyword** option of **FractGen** *i*n the command-line of the *condor_cod_activate* command.

NOTE: If a user is defining all attributes of their COD application in the Condor configuration files, and the *condor_startd* daemon on the COD resource they are using is running as root, the user must also define `Owner` to be the user that the COD application should run as (see section 4.3.3 above).

Defining Attributes with the *condor_cod* Tool

COD users may define attributes dynamically (at the time they spawn a COD application). In this case, the user writes the ClassAd attributes into a file, and the file name is passed to the *condor_cod_activate* tool using the **-jobad** command-line option. These attributes are read by the *condor_cod* tool and passed through the system onto the *condor_starter* daemon which spawns the COD application. If the file name given is `-`, the *condor_cod* tool will read from standard input (`stdin`).

Users should not add a keyword prefix when defining attributes with the *condor_cod_activate* tool. The attribute names can be used in the file directly.

WARNING: The current syntax for this file is not the same as the syntax in the file used with *condor_submit*.

NOTE: Users should not define the `Owner` attribute when using *condor_cod_activate* on the command line, since Condor will automatically insert the correct value based on what user runs the *condor_cod_activate* command and how that user authenticates to the COD resource. If a user defines an attribute that does not match the authenticated identity, Condor treats this case as an error, and it will fail to launch the application.

4.3.4 Managing COD Resource Claims

Separate commands are provided by Condor to manage COD claims on batch resources. Once created, each COD claim has a unique identifying string, called the claim ID. Most commands require a claim ID to specify which claim you wish to act on. These commands are the means by which COD applications interact with the rest of the Condor system. They should be issued by the controller application to manage its compute nodes. Here is a list of the commands:

Request Create a new COD claim on a given resource.

Activate Spawn a specific application on a specific COD claim.

Suspend Suspend a running application within a specific COD claim.

Renew Renew the lease to a COD claim.

Resume Resume a suspended application on a specific COD claim.

Deactivate Shut down an application, but hold onto the COD claim for future use.

Release Destroy a specific COD claim, and shut down any job that is currently running on it.

Delegate proxy Send an x509 proxy credential to the specific COD claim (optional, only required in rare cases like using glexec to spawn the *condor_starter* at the execute machine where the COD job is running).

To issue these commands, a user or application invokes the *condor_cod* tool. A command may be specified as the first argument to this tool, as

```
condor_cod request -name c02.cs.wisc.edu
```

or the *condor_cod* tool can be installed in such a way that the same binary is used for a set of names, as

```
condor_cod_request -name c02.cs.wisc.edu
```

Other than the command name itself (which must be included in full) additional options supported by each tool can be abbreviated to the shortest unambiguous value. For example, **-name** can also be specified as **-n**. However, for a command like *condor_cod_activate* that supports both **-classad** and **-cluster**, the user must use at least **-cla** or **-clu**. If the user specifies an ambiguous option, the *condor_cod* tool will exit with an error message.

In addition, there is now a **-cod** option to *condor_status*.

The following sections describe each option in greater detail.

Request

A user must be granted authorization to create COD claims on a specific machine. In addition, when the user uses these COD claims, the application binary or script they wish to run (and any input data) must be pre-staged on the machine. Therefore, a user cannot simply request a COD claim at random.

The user specifies the resource on which to make a COD claim. This is accomplished by specifying the name of the *condor_startd* daemon desired by invoking *condor_cod_request* with the **-name** option and the resource name (usually the host name). For example:

```
condor_cod_request -name c02.cs.wisc.edu
```

If the *condor_startd* daemon desired belongs to a different Condor pool than the one where executing the COD commands, use the **-pool** option to provide the name of the central manager machine of the other pool. For example:

```
condor_cod_request -name c02.cs.wisc.edu -pool condor.cs.wisc.edu
```

An alternative is to provide the IP address and port number where the *condor_startd* daemon is listening with the **-addr** option. This information can be found in the *condor_startd* ClassAd as the attribute `StartdIpAddr` or by reading the log file when the *condor_startd* first starts up. For example:

```
condor_cod_request -addr "<128.105.146.102:40967>"
```

If neither **-name** or **-addr** are specified, *condor_cod_request* attempts to connect to the *condor_startd* daemon running on the local machine (where the request command was issued).

If the *condor_startd* daemon to be used for the COD claim is an SMP machine and has multiple slots, specify which resource on the machine to use for COD by providing the full name of the resource, not just the host name. For example:

```
condor_cod_request -name slot2@c02.cs.wisc.edu
```

A constraint on what slot is desired may be provided, instead of specifying it by name. For example, to run on machine c02.cs.wisc.edu, not caring which slot is used, so long as it the machine is not currently running a job, use something like:

```
condor_cod_request -name c02.cs.wisc.edu -requirements 'State!="Claimed"'
```

In general, be careful with shell quoting issues, so that your shell is not confused by the ClassAd expression syntax (in particular if the expression includes a string). The safest method is to enclose any requirement expression within single quote marks (as shown above).

Once a given *condor_startd* daemon has been contacted to request a new COD claim, the *condor_startd* daemon checks for proper authorization of the user issuing the command. If the user has the authority, and the *condor_startd* daemon finds a resource that matches any given requirements, the *condor_startd* daemon creates a new COD claim and gives it a unique identifier, the claim ID. This ID is used to identify COD claims when using other commands. If *condor_cod_request* succeeds, the claim ID for the new claim is printed out to the screen. All other commands to manage this claim require the claim ID to be provided as a command-line option.

When the *condor_startd* daemon assigns a COD claim, the ClassAd describing the resource is returned to the user that requested the claim. This ClassAd is a snap-shot of the output of `condor_status -long` for the given machine. If *condor_cod_request* is invoked with the **-classad** option (which takes a file name as an argument), this ClassAd will be written out to the given file. Otherwise, the ClassAd is printed to the screen. The only essential piece of information in this ClassAd is the Claim ID, so that is printed to the screen, even if the whole ClassAd is also being written to a file.

The claim ID as given after listing the machine ClassAd appears as this example:

```
ID of new claim is: "<128.105.121.21:49973>#1073352104#4"
```

When using this claim ID in further commands, include the quote marks as well as all the characters in between the quote marks.

NOTE: Once a COD claim is created, there is no persistent record of it kept by the *condor_startd* daemon. So, if the *condor_startd* daemon is restarted for any reason, all existing COD claims will be destroyed and the new *condor_startd* daemon will not recognize any attempts to use the previous claims.

Also note that it is your responsibility to ensure that the claim is eventually removed (see section 4.3.4). Failure to remove the COD claim will result in the *condor_startd* continuing to hold a record of the claim for as long as *condor_startd* continues running. If a very large number of such claims are accumulated by the *condor_startd*, this can impact its performance. Even worse: if a COD claim is unintentionally left in an activated state, this results in the suspension of any batch job running on the same resource for as long as the claim remains activated. For this reason, an optional **-lease** argument is supported by *condor_cod_request*. This tells the *condor_startd* to automatically release the COD claim after the specified number of seconds unless the lease is renewed with *condor_cod_renew*. The default lease is infinitely long.

Activate

Once a user has created a valid COD claim and has the claim ID, the next step is to spawn a COD job using the claim. The way to do this is to activate the claim, using the *condor_cod_activate* command. Once a COD application is active on a COD claim, the COD claim will move into the `Running` state, and any batch Condor job on the same resource will be suspended. Whenever the COD application is inactive (either suspended, removed from the machine, or if it exits on its own), the state of the COD claim changes. The new state depends on why the application became inactive.

The batch Condor job then resumes.

To activate a COD claim, first define attributes about the job to be run in either the local configuration of the COD resource, or in a separate file as described in this manual section. Invoke the *condor_cod_activate* command to launch a specific instance of the job on a given COD claim ID. The options given to *condor_cod_activate* vary depending on if the job attributes are defined in the configuration file or are passed via a file to the *condor_cod_activate* tool itself. However, the **-id** option is always required by *condor_cod_activate*, and this option should be followed by a COD claim ID that the user acquired via *condor_cod_request*.

If the application is defined in the configuration files for the COD resource, the user provides the keyword (described in section 4.3.3) that uniquely identifies the application's configuration attributes. To continue the example from that section, the user would spawn their job by specifying `-keyword FractGen`, for example:

```
condor_cod_activate -id "<claim_id>" -keyword FractGen
```

Substitute the `<claim_id>` with the valid Cod Claim Id. Using the same example as given above, this example would be:

```
condor_cod_activate -id "<128.105.121.21:49973>#1073352104#4" -keyword FractGen
```

If the job attributes are placed into a file to be passed to the *condor_cod_activate* tool, the user must provide the name of the file using the **-jobad** option. For example, if the job attributes were defined in a file named `cod-fractgen.txt`, the user spawns the job using the command:

```
condor_cod_activate -id "<claim_id>" -jobad cod-fractgen.txt
```

Alternatively, if the filename specified with **-jobad** is `-`, the *condor_cod_activate* tool reads the job ClassAd from standard input (`stdin`).

Regardless of how the job attributes are defined, there are other options that *condor_cod_activate* accepts. These options specify the job ID for the application to be run. The job ID can either be specified in the job's ClassAd, or it can be specified on the command line to *condor_cod_activate*. These options are **-cluster** and **-proc**. For example, to launch a COD job with keyword `foo` as cluster 23, proc 5, or 23.5, the user invokes:

```
condor_cod_activate -id "<claim_id>" -key foo -cluster 23 -proc 5
```

The **-cluster** and **-proc** arguments are optional, since the job ID is not required for COD. If not specified, the job ID defaults to `1.0`.

Suspend

Once a COD application has been activated with *condor_cod_activate* and is running on a COD resource, it may be temporarily suspended using *condor_cod_suspend*. In this case, the claim state

becomes `Suspended`. Once a given COD job is suspended, if there are no other running COD jobs on the resource, a Condor batch job can use the resource. By suspending the COD application, the batch job is allowed to run. If a resource is idle when a COD application is first spawned, suspension of the COD job makes the batch resource available for use in the Condor system. Therefore, whenever a COD application has no work to perform, it should be suspended to prevent the resource from being wasted.

The interface of *condor_cod_suspend* supports the single option **-id**, to specify the COD claim ID to be suspended. For example:

```
condor_cod_suspend -id "<claim_id>"
```

If the user attempts to suspend a COD job that is not running, *condor_cod_suspend* exits with an error message. The COD job may not be running because it is already suspended or because the job was never spawned on the given COD claim in the first place.

Renew

This command tells the *condor_startd* to renew the lease on the COD claim for the amount of lease time specified when the claim was created. See section 4.3.4 for more information on using leases.

The *condor_cod_renew* tool supports only the **-id** option to specify the COD claim ID the user wishes to renew. For example:

```
condor_cod_renew -id "<claim_id>"
```

If the user attempts to renew a COD job that no longer exists, *condor_cod_renew* exits with an error message.

Resume

Once a COD application has been suspended with *condor_cod_suspend*, it can be resumed using *condor_cod_resume*. In this case, the claim state returns to `Running`. If there is a regular batch job running on the same resource, it will automatically be suspended if a COD application is resumed.

The *condor_cod_resume* tool supports only the **-id** option to specify the COD claim ID the user wishes to resume. For example:

```
condor_cod_resume -id "<claim_id>"
```

If the user attempts to resume a COD job that is not suspended, *condor_cod_resume* exits with an error message.

Deactivate

If a given COD application does not exit on its own and needs to be removed manually, invoke the *condor_cod_deactivate* command to kill the job, but leave the COD claim ID valid for future COD jobs. The user must specify the claim ID they wish to deactivate using the **-id** option. For example:

```
condor_cod_deactivate -id "<claim_id>"
```

By default, *condor_cod_deactivate* attempts to gracefully cleanup the COD application and give it time to exit. In this case the COD claim goes into the `Vacating` state and the *condor_starter* process controlling the job will send it the `KillSig` defined for the job (SIGTERM by default). This allows the COD job to catch the signal and do whatever final work is required to exit cleanly.

However, if the program is stuck or if the user does not want to give the application time to clean itself up, the user may use the **-fast** option to tell the *condor_starter* to quickly kill the job and all its descendants using SIGKILL. In this case the COD claim goes into the `Killing` state. For example:

```
condor_cod_deactivate -id "<claim_id>" -fast
```

In either case, once the COD job has finally exited, the COD claim will go into the `Idle` state and will be available for future COD applications. If there are no other active COD jobs on the same resource, the resource would become available for batch Condor jobs. Whenever the user wishes to spawn another COD application, they can reuse this idle COD claim by using the same claim ID, without having to go through the process of running *condor_cod_request*.

If the user attempts a *condor_cod_deactivate* request on a COD claim that is neither `Running` nor `Suspended`, the *condor_cod* tool exits with an error message.

Release

If users no longer wish to use a given COD claim, they can release the claim with the *condor_cod_release* command. If there is a COD job running on the claim, the job will first be shut down (as if *condor_cod_deactivate* was used), and then the claim itself is removed from the resource and the claim ID is destroyed. Further attempts to use the claim ID for any COD commands will fail.

The *condor_cod_release* command always prints out the state the COD claim was in when the request was received. This way, users can know what state a given COD application was in when the claim was destroyed.

Like most COD commands, *condor_cod_release* requires the claim ID to be specified using **-id**. In addition, *condor_cod_release* supports the **-fast** option (described above in the section about *condor_cod_deactivate*). If there is a job running or suspended on the claim when it is released with `condor_cod_release -fast`, the job will be immediately killed. If **-fast** is not specified, the

default behavior is to use a graceful shutdown, sending whatever signal is specified in the `KillSig` attribute for the job (SIGTERM by default).

Delegate proxy

In some cases, a user will want to delegate a copy of their user credentials (in the form of an x509 proxy) to the machine where one of their COD jobs will run. For example, sites wishing to spawn the *condor_starter* using glexec will need a copy of this credential before the claim can be activated. Therefore, beginning with Condor version 6.9.2, COD users have access to a the command `delegate_proxy`. If users do not specifically require this proxy delegation, this command should not be used and the rest of this section can be skipped.

The `delegate_proxy` command optionally takes a **-x509proxy** argument to specify the path to the proxy file to use. Otherwise, it uses the same discovery logic that *condor_submit* uses to find the user's currently active proxy.

Just like every other COD command (except `request`), this command requires a valid COD claim id (specified with **-id**) to indicate what COD claim you wish to delegate the credentials to.

This command can only be sent to idle COD claims, so it should be done before `activate` is run for the first time. However, once a proxy has been delegated, it can be reused by successive claim activations, so normally this step only has to happen once, not before every activate. If a proxy is going to expire, and a new one should be sent, this should only happen after the existing COD claim has been deactivated.

4.3.5 Limitations of COD Support in Condor

Condor's support for COD has a few limitations.

The following items are all limitations we plan to remove in future releases of Condor:

- Applications and data must be pre-staged at a given machine.
- There is no way to define limits for how long a given COD claim can be active, how often it is run, and so on.
- There is no accounting done for applications run under COD claims. Therefore, use of a lot of COD resources in a given Condor pool does not adversely affect user priority.

None of the above items are fundamentally difficult to add and we hope to address them relatively quickly. If you run into one of these limitations, and it is a barrier to using COD, please contact condor-admin@cs.wisc.edu with the subject "COD limitation" to gain quick help.

The following list are more fundamental limitations that we do not plan to address:

- COD claims are not persistent on a given *condor_startd* daemon.
- Condor does not provide a mechanism to parallelize a graphic application to take advantage of COD. The Condor Team is not in the business of developing applications, we only provide mechanisms to execute them.

4.4 Hooks

A *hook* is an external program or script invoked by Condor.

Job hooks that fetch work allow sites to write their own programs or scripts, and allow Condor to invoke these hooks at the right moments to accomplish the desired outcome. This eliminates the expense of the matchmaking and scheduling provided by the *condor_schedd* and the *condor_negotiator*, although at the price of the flexibility they offer. Therefore, job hooks that fetch work allow Condor to more easily and directly interface with external scheduling systems.

Hooks may also behave as a Job Router.

The Daemon ClassAd hooks permit the *condor_startd* and the *condor_schedd* daemons to execute hooks once or on a periodic basis.

4.4.1 Job Hooks That Fetch Work

In the past, Condor has always sent work to the execute machines by pushing jobs to the *condor_startd* daemon, either from the *condor_schedd* daemon or via *condor_cod*. Beginning with the Condor version 7.1.0, the *condor_startd* daemon now has the ability to pull work by fetching jobs via a system of plug-ins or hooks. Any site can configure a set of hooks to fetch work, completely outside of the usual Condor matchmaking system.

A projected use of the hook mechanism implements what might be termed a *glide-in factory*, especially where the factory is behind a firewall. Without using the hook mechanism to fetch work, a glide-in *condor_startd* daemon behind a firewall depends on CCB or GCB to help it listen and eventually receive work pushed from elsewhere. With the hook mechanism, a glide-in *condor_startd* daemon behind a firewall uses the hook to pull work. The hook needs only an outbound network connection to complete its task, thereby being able to operate from behind the firewall, without the intervention of CCB or GCB.

Periodically, each execution slot managed by a *condor_startd* will invoke a hook to see if there is any work that can be fetched. Whenever this hook returns a valid job, the *condor_startd* will evaluate the current state of the slot and decide if it should start executing the fetched work. If the slot is unclaimed and the `Start` expression evaluates to `True`, a new claim will be created for the fetched job. If the slot is claimed, the *condor_startd* will evaluate the `Rank` expression relative to the fetched job, compare it to the value of `Rank` for the currently running job, and decide if the existing job should be preempted due to the fetched job having a higher rank. If the slot is unavailable for

whatever reason, the *condor_startd* will refuse the fetched job and ignore it. Either way, once the *condor_startd* decides what it should do with the fetched job, it will invoke another hook to reply to the attempt to fetch work, so that the external system knows what happened to that work unit.

If the job is accepted, a claim is created for it and the slot moves into the Claimed state. As soon as this happens, the *condor_startd* will spawn a *condor_starter* to manage the execution of the job. At this point, from the perspective of the *condor_startd*, this claim is just like any other. The usual policy expressions are evaluated, and if the job needs to be suspended or evicted, it will be. If a higher-ranked job being managed by a *condor_schedd* is matched with the slot, that job will preempt the fetched work.

The *condor_starter* itself can optionally invoke additional hooks to help manage the execution of the specific job. There are hooks to prepare the execution environment for the job, periodically update information about the job as it runs, notify when the job exits, and to take special actions when the job is being evicted.

Assuming there are no interruptions, the job completes, and the *condor_starter* exits, the *condor_startd* will invoke the hook to fetch work again. If another job is available, the existing claim will be reused and a new *condor_starter* is spawned. If the hook returns that there is no more work to perform, the claim will be evicted, and the slot will return to the Owner state.

Work Fetching Hooks Invoked by Condor

There are a handful of hooks invoked by Condor related to fetching work, some of which are called by the *condor_startd* and others by the *condor_starter*. Each hook is described, including when it is invoked, what task it is supposed to accomplish, what data is passed to the hook, what output is expected, and, when relevant, the exit status expected.

Hook: Fetch Work The hook defined by the configuration variable `<Keyword>_HOOK_FETCH_WORK` is invoked whenever the *condor_startd* wants to see if there is any work to fetch. There is a related configuration variable called `FetchWorkDelay` which determines how long the *condor_startd* will wait between attempts to fetch work, which is described in detail in within section 4.4.1 on page 514. `<Keyword>_HOOK_FETCH_WORK` is the most important hook in the whole system, and is the only hook that must be defined for any of the other *condor_startd* hooks to operate.

The job ClassAd returned by the hook needs to contain enough information for the *condor_starter* to eventually spawn the work. The required and optional attributes in this ClassAd are identical to the ones described for Computing on Demand (COD) jobs in section 4.3.3 on COD Application Attributes, page 498.

Command-line arguments passed to the hook None.

Standard input given to the hook ClassAd of the slot that is looking for work.

Expected standard output from the hook ClassAd of a job that can be run. If there is no work, the hook should return no output.

Exit status of the hook Ignored.

Hook: Reply Fetch The hook defined by the configuration variable `<Keyword>_HOOK_REPLY_FETCH` is invoked whenever `<Keyword>_HOOK_FETCH_WORK` returns data and the *condor_startd* decides if it is going to accept the fetched job or not.

The *condor_startd* will not wait for this hook to return before taking other actions, and it ignores all output. The hook is simply advisory, and it has no impact on the behavior of the *condor_startd*.

Command-line arguments passed to the hook Either the string `accept` or `reject`.

Standard input given to the hook A copy of the job ClassAd and the slot ClassAd (separated by the string `-----` and a new line).

Expected standard output from the hook None.

Exit status of the hook Ignored.

Hook: Evict Claim The hook defined by the configuration variable `<Keyword>_HOOK_EVICT_CLAIM` is invoked whenever the *condor_startd* needs to evict a claim representing fetched work.

The *condor_startd* will not wait for this hook to return before taking other actions, and ignores all output. The hook is simply advisory, and has no impact on the behavior of the *condor_startd*.

Command-line arguments passed to the hook None.

Standard input given to the hook A copy of the job ClassAd and the slot ClassAd (separated by the string `-----` and a new line).

Expected standard output from the hook None.

Exit status of the hook Ignored.

Hook: Prepare Job The hook defined by the configuration variable `<Keyword>_HOOK_PREPARE_JOB` is invoked by the *condor_starter* before a job is going to be run. This hook provides a chance to execute commands to set up the job environment, for example, to transfer input files.

The *condor_starter* waits until this hook returns before attempting to execute the job. If the hook returns a non-zero exit status, the *condor_starter* will assume an error was reached while attempting to set up the job environment and abort the job.

Command-line arguments passed to the hook None.

Standard input given to the hook A copy of the job ClassAd.

Expected standard output from the hook A set of attributes to insert or update into the job ad. For example, changing the `Cmd` attribute to a quoted string changes the executable to be run.

Exit status of the hook 0 for success preparing the job, any non-zero value on failure.

Hook: Update Job Info The hook defined by the configuration variable `<Keyword>_HOOK_UPDATE_JOB_INFO` is invoked periodically during the life of the job to update information about the status of the job. When the job is first spawned, the *condor_starter* will invoke this hook after

STARTER_INITIAL_UPDATE_INTERVAL seconds (defaults to 8). Thereafter, the *condor_starter* will invoke the hook every STARTER_UPDATE_INTERVAL seconds (defaults to 300, which is 5 minutes).

The *condor_starter* will not wait for this hook to return before taking other actions, and ignores all output. The hook is simply advisory, and has no impact on the behavior of the *condor_starter*.

Command-line arguments passed to the hook None.

Standard input given to the hook A copy of the job ClassAd that has been augmented with additional attributes describing the current status and execution behavior of the job.

The additional attributes included inside the job ClassAd are:

JobState The current state of the job. Can be either "Running" or "Suspended".

JobPid The process identifier for the initial job directly spawned by the *condor_starter*.

NumPids The number of processes that the job has currently spawned.

JobStartDate The epoch time when the job was first spawned by the *condor_starter*.

RemoteSysCpu The total number of seconds of system CPU time (the time spent at system calls) the job has used.

RemoteUserCpu The total number of seconds of user CPU time the job has used.

ImageSize The memory image size of the job in Kbytes.

Expected standard output from the hook None.

Exit status of the hook Ignored.

Hook: Job Exit The hook defined by the configuration variable <Keyword>_HOOK_JOB_EXIT is invoked by the *condor_starter* whenever a job exits, either on its own or when being evicted from an execution slot.

The *condor_starter* will wait for this hook to return before taking any other actions. In the case of jobs that are being managed by a *condor_shadow*, this hook is invoked before the *condor_starter* does its own optional file transfer back to the submission machine, writes to the local user log file, or notifies the *condor_shadow* that the job has exited.

Command-line arguments passed to the hook A string describing how the job exited:

- exit The job exited or died with a signal on its own.
- remove The job was removed with *condor_rm* or as the result of user job policy expressions (for example, PeriodicRemove).
- hold The job was held with *condor_hold* or the user job policy expressions (for example, PeriodicHold).
- evict The job was evicted from the execution slot for any other reason (PREEMPT evaluated to TRUE in the *condor_startd*, *condor_vacate*, *condor_off*, etc).

Standard input given to the hook A copy of the job ClassAd that has been augmented with additional attributes describing the execution behavior of the job and its final results.

The job ClassAd passed to this hook contains all of the extra attributes described above for `<Keyword>_HOOK_UPDATE_JOB_INFO`, and the following additional attributes that are only present once a job exits:

`ExitReason` A human-readable string describing why the job exited.

`ExitBySignal` A boolean indicating if the job exited due to being killed by a signal, or if it exited with an exit status.

`ExitSignal` If `ExitBySignal` is true, the signal number that killed the job.

`ExitCode` If `ExitBySignal` is false, the integer exit code of the job.

`JobDuration` The number of seconds that the job ran during this invocation.

Expected standard output from the hook None.

Exit status of the hook Ignored.

Keywords to Define Job Fetch Hooks in the Condor Configuration files

Hooks are defined in the Condor configuration files by prefixing the name of the hook with a keyword. This way, a given machine can have multiple sets of hooks, each set identified by a specific keyword.

Each slot on the machine can define a separate keyword for the set of hooks that should be used with `SLOT<N>_JOB_HOOK_KEYWORD`. For example, on slot 1, the variable name will be called `SLOT1_JOB_HOOK_KEYWORD`. If the slot-specific keyword is not defined, the *condor_startd* will use a global keyword as defined by `STARTD_JOB_HOOK_KEYWORD`.

Once a job is fetched via `<Keyword>_HOOK_FETCH_WORK`, the *condor_startd* will insert the keyword used to fetch that job into the job ClassAd as `HookKeyword`. This way, the same keyword will be used to select the hooks invoked by the *condor_starter* during the actual execution of the job. However, the `STARTER_JOB_HOOK_KEYWORD` can be defined to force the *condor_starter* to always use a given keyword for its own hooks, instead of looking the job ClassAd for a `HookKeyword` attribute.

For example, the following configuration defines two sets of hooks, and on a machine with 4 slots, 3 of the slots use the global keyword for running work from a database-driven system, and one of the slots uses a custom keyword to handle work fetched from a web service.

```
# Most slots fetch and run work from the database system.
STARTD_JOB_HOOK_KEYWORD = DATABASE

# Slot4 fetches and runs work from a web service.
SLOT4_JOB_HOOK_KEYWORD = WEB

# The database system needs to both provide work and know the reply
# for each attempted claim.
DATABASE_HOOK_DIR = /usr/local/condor/fetch/database
DATABASE_HOOK_FETCH_WORK = $(DATABASE_HOOK_DIR)/fetch_work.php
DATABASE_HOOK_REPLY_FETCH = $(DATABASE_HOOK_DIR)/reply_fetch.php
```

```
# The web system only needs to fetch work.
WEB_HOOK_DIR = /usr/local/condor/fetch/web
WEB_HOOK_FETCH_WORK = $(WEB_HOOK_DIR)/fetch_work.php
```

The keywords "DATABASE" and "WEB" are completely arbitrary, so each site is encouraged to use different (more specific) names as appropriate for their own needs.

Defining the FetchWorkDelay Expression

There are two events that trigger the *condor_startd* to attempt to fetch new work:

- the *condor_startd* evaluates its own state
- the *condor_starter* exits after completing some fetched work

Even if a given compute slot is already busy running other work, it is possible that if it fetched new work, the *condor_startd* would prefer this newly fetched work (via the Rank expression) over the work it is currently running. However, the *condor_startd* frequently evaluates its own state, especially when a slot is claimed. Therefore, administrators can define a configuration variable which controls how long the *condor_startd* will wait between attempts to fetch new work. This variable is called FetchWorkDelay .

The FetchWorkDelay expression must evaluate to an integer, which defines the number of seconds since the last fetch attempt completed before the *condor_startd* will attempt to fetch more work. However, as a ClassAd expression (evaluated in the context of the ClassAd of the slot considering if it should fetch more work, and the ClassAd of the currently running job, if any), the length of the delay can be based on the current state the slot and even the currently running job.

For example, a common configuration would be to always wait 5 minutes (300 seconds) between attempts to fetch work, unless the slot is Claimed/Idle, in which case the *condor_startd* should fetch immediately:

```
FetchWorkDelay = ifThenElse(State == "Claimed" && Activity == "Idle", 0, 300)
```

If the *condor_startd* wants to fetch work, but the time since the last attempted fetch is shorter than the current value of the delay expression, the *condor_startd* will set a timer to fetch as soon as the delay expires.

If this expression is not defined, the *condor_startd* will default to a five minute (300 second) delay between all attempts to fetch work.

Example Hook: Specifying the Executable at Execution Time

The availability of multiple versions of an application leads to the need to specify one of the versions. As an example, consider that the java universe utilizes a single, fixed JVM. There may be multiple

JVMs available, and the Condor job may need to make the choice of JVM version. The use of a job hook solves this problem. The job does not use the java universe, and instead uses the vanilla universe in combination with a prepare job hook to overwrite the `Cmd` attribute of the job ClassAd. This attribute is the name of the executable the *condor_starter* daemon will invoke, thereby selecting the specific JVM installation.

In the configuration of the execute machine:

```
JAVA5_HOOK_PREPARE_JOB = $(LIBEXEC)/java5_prepare_hook
```

With this configuration, a job that sets the `HookKeyword` attribute with

```
+HookKeyword = "JAVA5"
```

in the submit description file causes the *condor_starter* will run the hook specified by `JAVA5_HOOK_PREPARE_JOB` before running this job. Note that the double quote marks are required to correctly define the attribute. Any output from this hook is an update to the job ClassAd. Therefore, the hook that changes the executable may be

```
#!/bin/sh

# Read and discard the job ClassAd
cat > /dev/null
echo 'Cmd = "/usr/java/java5/bin/java"'
```

The submit description file for this example job may be

```
universe = vanilla
executable = /usr/bin/java
arguments = Hello
# match with a machine that advertises the JAVA5 hook
requirements = (JAVA5_HOOK_PREPARE_JOB =!= UNDEFINED)

should_transfer_files = always
when_to_transfer_output = on_exit
transfer_input_files = Hello.class

output = hello.out
error  = hello.err
log    = hello.log

+HookKeyword="JAVA5"
queue
```

Note that the **requirements** command ensures that this job matches with a machine that has `JAVA5_HOOK_PREPARE_JOB` defined.

4.4.2 Hooks for a Job Router

Job Router Hooks allow for an alternate transformation and/or monitoring than the *condor_job_router* daemon implements. Routing is still managed by the *condor_job_router* daemon, but if the Job Router Hooks are specified, then these hooks will be used to transform and monitor the job instead.

Job Router Hooks are similar in concept to Fetch Work Hooks, but they are limited in their scope. A hook is an external program or script invoked by the *condor_job_router* daemon at various points during the life cycle of a routed job.

The following sections describe how and when these hooks are used, what hooks are invoked at various stages of the job's life, and how to configure Condor to use these Hooks.

Hooks Invoked for Job Routing

The Job Router Hooks allow for replacement of the transformation engine used by Condor for routing a job. Since the external transformation engine is not controlled by Condor, additional hooks provide a means to update the job's status in Condor, and to clean up upon exit or failure cases. This allows one job to be transformed to just about any other type of job that Condor supports, as well as to use execution nodes not normally available to Condor.

It is important to note that if the Job Router Hooks are utilized, then Condor will not ignore or work around a failure in any hook execution. If a hook is configured, then Condor assumes its invocation is required and will not continue by falling back to a part of its internal engine. For example, if there is a problem transforming the job using the hooks, Condor will not fall back on its transformation accomplished without the hook to process the job.

There are 2 ways in which the Job Router Hooks may be enabled. A job's submit description file may cause the hooks to be invoked with

```
+HookKeyword = "HOOKNAME"
```

Adding this attribute to the job's ClassAd causes the *condor_job_router* daemon on the submit machine to invoke hooks prefixed with the defined keyword. `HOOKNAME` is a string chosen as an example; any string may be used.

The job's ClassAd attribute definition of `HookKeyword` takes precedence, but if not present, hooks may be enabled by defining on the submit machine the configuration variable

```
JOB_ROUTER_HOOK_KEYWORD = HOOKNAME
```

Like the example attribute above, `HOOKNAME` represents a chosen name for the hook, replaced as desired or appropriate.

There are 4 hooks that the Job Router can be configured to use. Each hook will be described below along with data passed to the hook and expected output. All hooks must exit successfully.

Hook: Translate The hook defined by the configuration variable `<Keyword>_HOOK_TRANSLATE_JOB` is invoked when the Job Router has determined that a job meets the definition for a route. This hook is responsible for doing the transformation of the job and configuring any resources that are external to Condor if applicable.

Command-line arguments passed to the hook None.

Standard input given to the hook The first line will be the route that the job matched as defined in Condor's configuration files followed by the job ClassAd, separated by the string `"------"` and a new line.

Expected standard output from the hook The transformed job.

Exit status of the hook 0 for success, any non-zero value on failure.

Hook: Update Job Info The hook defined by the configuration variable `<Keyword>_HOOK_UPDATE_JOB_INFO` is invoked to provide status on the specified routed job when the Job Router polls the status of routed jobs at intervals set by `JOB_ROUTER_POLLING_PERIOD`.

Command-line arguments passed to the hook None.

Standard input given to the hook The routed job ClassAd that is to be updated.

Expected standard output from the hook The job attributes to be updated in the routed job, or nothing, if there was no update. To prevent clashing with Condor's management of job attributes, only attributes that are not managed by Condor should be output from this hook.

Exit status of the hook 0 for success, any non-zero value on failure.

Hook: Job Finalize The hook defined by the configuration variable `<Keyword>_HOOK_JOB_FINALIZE` is invoked when the Job Router has found that the job has completed. Any output from the hook is treated as an update to the source job.

Command-line arguments passed to the hook None.

Standard input given to the hook The source job ClassAd, followed by the routed copy Classad that completed, separated by the string `"------"` and a new line.

Expected standard output from the hook An updated source job ClassAd, or nothing if there was no update.

Exit status of the hook 0 for success, any non-zero value on failure.

Hook: Job Cleanup The hook defined by the configuration variable `<Keyword>_HOOK_JOB_CLEANUP` is invoked when the Job Router finishes managing the job. This hook will be invoked regardless of whether the job completes successfully or not, and must exit successfully.

Command-line arguments passed to the hook None.

Standard input given to the hook The job ClassAd that the Job Router is done managing.

Expected standard output from the hook None.

Exit status of the hook 0 for success, any non-zero value on failure.

4.4.3 Daemon ClassAd Hooks

The *Daemon ClassAd Hook* mechanism is used to run executables (called jobs) directly from the *condor_startd* and *condor_schedd* daemons. The output from these jobs is incorporated into the machine ClassAd generated by the respective daemon. The mechanism and associated jobs have been identified by various names, including the *Startd Cron*, dynamic attributes, and a distribution of executables collectively known as *Hawkeye*.

Pool management tasks can be enhanced by using a daemon's ability to periodically run executables. The executables are expected to generate ClassAd attributes as their output, which are incorporated into the machine ClassAd. Policy expressions may then reference the dynamic attributes.

Configuration variables related to Daemon ClassAd Hooks are defined within section 3.3.37. Here is a complete configuration example. It defines all three of the available types of jobs: ones that use the *condor_startd*, benchmark jobs, and ones that use the *condor_schedd*.

```
#
# Startd Cron Stuff
#
# auxiliary variable to use in identifying locations of files
MODULES = $(ROOT)/modules

STARTD_CRON_CONFIG_VAL = $(RELEASE_DIR)/bin/condor_config_val
STARTD_CRON_MAX_JOB_LOAD = 0.2
STARTD_CRON_JOBLIST =

# Test job
STARTD_CRON_JOBLIST = $(STARTD_CRON_JOBLIST) test
STARTD_CRON_TEST_MODE = OneShot
STARTD_CRON_TEST_RECONFIG_RERUN = True
STARTD_CRON_TEST_PREFIX = test_
STARTD_CRON_TEST_EXECUTABLE = $(MODULES)/test
STARTD_CRON_TEST_KILL = True
STARTD_CRON_TEST_PARAM0 = abc
STARTD_CRON_TEST_PARAM1 = 123
STARTD_CRON_TEST_SLOTS = 1
STARTD_CRON_TEST_JOB_LOAD = 0.01

# job 'date'
STARTD_CRON_JOBLIST = $(STARTD_CRON_JOBLIST) date
STARTD_CRON_DATE_MODE = Periodic
STARTD_CRON_DATE_EXECUTABLE = $(MODULES)/date
STARTD_CRON_DATE_PERIOD = 15s
STARTD_CRON_DATE_JOB_LOAD = 0.01

# Job 'foo'
STARTD_CRON_JOBLIST = $(STARTD_CRON_JOBLIST) foo
STARTD_CRON_FOO_EXECUTABLE = $(MODULES)/foo
STARTD_CRON_FOO_PREFIX = Foo
STARTD_CRON_FOO_MODE = Periodic
STARTD_CRON_FOO_PERIOD = 10m
STARTD_CRON_FOO_JOB_LOAD = 0.2

#
```

```
# Benchmark Stuff
#
BENCHMARKS_JOBLIST = mips kflops

# MIPS benchmark
BENCHMARKS_MIPS_EXECUTABLE = $(LIBEXEC)/condor_mips
BENCHMARKS_MIPS_JOB_LOAD = 1.0

# KFLOPS benchmark
BENCHMARKS_KFLOPS_EXECUTABLE = $(LIBEXEC)/condor_kflops
BENCHMARKS_KFLOPS_JOB_LOAD = 1.0

#
# Schedd Cron Stuff
#
SCHEDD_CRON_CONFIG_VAL = $(RELEASE_DIR)/bin/condor_config_val
SCHEDD_CRON_JOBLIST =

# Test job
SCHEDD_CRON_JOBLIST = $(SCHEDD_CRON_JOBLIST) test
SCHEDD_CRON_TEST_MODE = OneShot
SCHEDD_CRON_TEST_RECONFIG_RERUN = True
SCHEDD_CRON_TEST_PREFIX = test_
SCHEDD_CRON_TEST_EXECUTABLE = $(MODULES)/test
SCHEDD_CRON_TEST_PERIOD = 5m
SCHEDD_CRON_TEST_KILL = True
SCHEDD_CRON_TEST_PARAM0 = abc
SCHEDD_CRON_TEST_PARAM1 = 123
```

4.5 Application Program Interfaces

4.5.1 Web Service

Condor's Web Service (WS) API provides a way for application developers to interact with Condor, without needing to utilize Condor's command-line tools. In keeping with the Condor philosophy of reliability and fault-tolerance, this API is designed to provide a simple and powerful way to interact with Condor. Condor daemons understand and implement the SOAP (Simple Object Access Protocol) XML API to provide a web service interface for Condor job submission and management.

To deal with the issues of reliability and fault-tolerance, a two-phase commit mechanism to provides a transaction-based protocol. The following API description describes interaction between a client using the API and both the *condor_schedd* and *condor_collector* daemons to illustrate transactions for use in job submission, queue management and ClassAd management functions.

Transactions

All applications using the API to interact with the *condor_schedd* will need to use transactions. A transaction is an ACID unit of work (atomic, consistent, isolated, and durable). The API limits the lifetime of a transaction, and both the client (application) and the server (the *condor_schedd* daemon) may place a limit on the lifetime. The server reserves the right to specify a maximum duration for a transaction.

The client initiates a transaction using the `beginTransaction()` method. It ends the transaction with either a commit (using `commitTransaction()`) or an abort (using `abortTransaction()`).

Not all operations in the API need to be performed within a transaction. Some accept a null transaction. A null transaction is a SOAP message with

```
<transaction xsi:type="ns1:Transaction" xsi:nil="true"/>
```

Often this is achieved by passing the programming language's equivalent of `null` in place of a transaction identifier. It is possible that some operations will have access to more information when they are used inside a transaction. For instance, a `getJobAds()`. query would have access to the jobs that are pending in a transaction, which are not committed and therefore not visible outside of the transaction. Transactions are as ACID compliant as possible. Therefore, do not query for information outside of a transaction on which to make a decision inside a transaction based on the query's results.

Job Submission

A ClassAd is required to describe a job. The job ClassAd will be submitted to the *condor_schedd* within a transaction using the `submit()` method. The complexity of job ClassAd creation may be simplified by the `createJobTemplate()` method. It returns an instance of a ClassAd structure that may be further modified. A necessary part of the job ClassAd are the job attributes `ClusterId` and `ProcId`, which uniquely identify the cluster and the job within a cluster. Allocation and assignment of (monotonically increasing) `ClusterId` values utilize the `newCluster()` method. Jobs may be submitted within the assigned cluster only until the `newCluster()` method is invoked a subsequent time. Each job is allocated and assigned a (monotonically increasing) `ProcId` within the current cluster using the `newJob()` method. Therefore, the sequence of method calls to submit a set of jobs initially calls `newCluster()`. This is followed by calls to `newJob()` and then `submit()` for each job within the cluster.

As an example, here are sample cluster and job numbers that result from the ordered calls to submission methods:

1. A call to `newCluster()`, assigns a `ClusterId` of 6.
2. A call to `newJob()`, assigns a `ProcId` of 0, as this is the first job within the cluster.

3. A call to `submit()` results in a job submission numbered 6.0.
4. A call to `newJob()`, assigns a `ProcId` of 1.
5. A call to `submit()` results in a job submission numbered 6.1.
6. A call to `newJob()`, assigns a `ProcId` of 2.
7. A call to `submit()` results in a job submission numbered 6.2.
8. A call to `newCluster()`, assigns a `ClusterId` of 7.
9. A call to `newJob()`, assigns a `ProcId` of 0, as this is the first job within the cluster.
10. A call to `submit()` results in a job submission numbered 7.0.
11. A call to `newJob()`, assigns a `ProcId` of 1.
12. A call to `submit()` results in a job submission numbered 7.1.

There is the potential that a call to `submit()` will fail. Failure means that the job is in the queue, and it typically indicates that something needed by the job has not been sent. As a result the job has no hope in successfully running. It is possible to recover from such a failure by trying to resend information that the job will need. It is also completely acceptable to abort and make another attempt. To simplify the client's effort in figuring out what the job requires, a `discoverJobRequirements()` method accepting a job ClassAd and returning a list of things that should be sent along with the job is provided.

File Transfer

A common job submission case requires the job's executable and input files to be transferred from the machine where the application is running to the machine where the *condor_schedd* daemon is running. This is the analogous situation to running *condor_submit* using the **-spool** or **-remote** option. The executable and input files must be sent directly to the *condor_schedd* daemon, which places all files in a spool location.

The two methods `declareFile()` and `sendFile()` work in tandem to transfer files to the *condor_schedd* daemon. The `declareFile()` method causes the *condor_schedd* daemon to create the file in its spool location, or indicate in its return value that the file already exists. This increases efficiency, as resending an existing file is a waste of resources. The `sendFile()` method sends base64 encoded data. `sendFile()` may be used to send an entire file, or chunks of files as desired.

The `declareFile()` method has both required and optional arguments. `declareFile()` requires the name of the file and its size in bytes. The optional arguments relate hash information. A hash type of `NOHASH` disables file verification; the *condor_schedd* daemon will not have a reliable way to determine the existence of the file being declared.

Methods for retrieving files are most useful when a job is completed. Consider the categorization of the typical life-cycle for a job:

Birth: The birth of a job begins with `submit()`.

Childhood: The job executes.

Middle Age: A completed job waits to be removed. As the job enters Middle Age, its `JobStatus` ClassAd attribute becomes Completed (the value 4).

Old Age: The job's information goes into the history log.

Once the job enters Middle Age, the `getFile()` method retrieves a file. The `listSpool()` method assists by providing a list of all the job's files in the spool location.

The job enters Old Age by the application's use of the `closeSpool()` method. It causes the *condor_schedd* daemon to remove the job from the queue, and the job's spool files are no longer available. As there is no requirement for the application to invoke the `closeSpool()` method, jobs can potentially remain in the queue forever. The configuration variable `SOAP_LEAVE_IN_QUEUE` may mitigate this problem. When this boolean variable evaluates to `False`, a job enters Old Age. A reasonable example for this configuration variable is

```
SOAP_LEAVE_IN_QUEUE = ((JobStatus==4) && ((ServerTime - CompletionDate) < (60 * 60 * 24)))
```

This expression results in Old age for a job (removed from the queue), once the job has been Middle Aged (been completed) for 24 hours.

Implementation Details

Condor daemons understand and communicate using the SOAP XML protocol. An application seeking to use this protocol will require code that handles the communication. The XML WSDL (Web Services Description Language) that Condor implements is included with the Condor distribution. It is in `$(RELEASE_DIR)/lib/webservice`. The WSDL must be run through a toolkit to produce language-specific routines that do communication. The application is compiled with these routines.

Condor must be configured to enable responses to SOAP calls. Please see section 3.3.33 for definitions of the configuration variables related to the web services API. The WS interface is listening on the *condor_schedd* daemon's command port. To obtain a list of all the the *condor_schedd* daemons in the pool with a WS interface, issue the command:

```
%  condor_status -schedd -constraint "HasSOAPInterface=?=TRUE"
```

With this information, a further command locates the port number to use:

```
% condor_status -schedd -constraint "HasSOAPInterface=?=TRUE" -l | grep MyAddress
```

Condor's security configuration must be set up such that access is authorized for the SOAP client. See Section 3.6.7 for information on how to set the `ALLOW_SOAP` and `DENY_SOAP` configuration variables.

The API's routines can be roughly categorized into ones that deal with

- Transactions
- Job Submission
- File Transfer
- Job Management
- ClassAd Management
- Version Information

The routines for each of these categories is detailed. Note that the signature provided will accurately reflect a routine's name, but that return values and parameter specification will vary according to the target programming language.

Get These Items Correct

- For jobs that are to be executed on Windows platforms, explicitly set the job ClassAd attribute `NTDomain`. This attribute defines the NT domain within which the job's owner authenticates. The attribute is necessary, and it is not set for the job by the `createJobTemplate()` function.

Methods for Transaction Management

`beginTransaction` Begin a transaction. A prototype is

```
StatusAndTransaction beginTransaction(int duration);
```

Parameters • `duration` The expected duration of the transaction.

Return Value If the function succeeds, the return value is `SUCCESS`; otherwise, see `StatusCode` for valid return values. Additionally, on success, the return value contains the new transaction.

`commitTransaction` Commits a transaction. A prototype is

```
Status commitTransaction(Transaction transaction);
```

Parameters • `transaction` The transaction to be committed.

Return Value If the function succeeds, the return value is `SUCCESS`; otherwise, see `StatusCode` for valid return values.

`abortTransaction` Abort a transaction. A prototype is

```
Status abortTransaction(Transaction transaction);
```

Parameters • `transaction` The transaction to be aborted.

Return Value If the function succeeds, the return value is `SUCCESS`; otherwise, see `StatusCode` for valid return values.

`extendTransaction` Request an extension in duration for a specific transaction. A prototype is

```
StatusAndTransaction extendTransaction( Transaction
transaction, int duration);
```

Parameters
- `transaction` The transaction to be extended.
- `duration` The duration of the extension.

Return Value If the function succeeds, the return value is `SUCCESS`; otherwise, see `StatusCode` for valid return values. Additionally, on success, the return value contains the transaction with the extended duration.

Methods for Job Submission

`submit` Submit a job. A prototype is

```
StatusAndRequirements submit(Transaction transaction, int
clusterId, int jobId, ClassAd jobAd);
```

Parameters
- `transaction` The transaction in which the submission takes place.
- `clusterId` The cluster identifier.
- `jobId` The job identifier.
- `jobAd` The ClassAd describing the job. Creation of this ClassAd can be simplified with `createJobTemplate();`.

Return Value If the function succeeds, the return value is `SUCCESS`; otherwise, see `StatusCode` for valid return values. Additionally, the return value contains the job's requirements.

`createJobTemplate` Request a job Class Ad, given some of the job requirements. This job Class Ad will be suitable for use when submitting the job. Note that the job attribute `NTDomain` is not set by this function, but must be set for jobs that will execute on Windows platforms. A prototype is

```
StatusAndClassAd createJobTemplate(int clusterId, int jobId,
String owner, UniverseType type, String command, String
arguments, String requirements);
```

Parameters
- `clusterId` The cluster identifier.
- `jobId` The job identifier.
- `owner` The name to be associated with the job.
- `type` The universe under which the job will run, where `type` can be one of the following:
  ```
  enum UniverseType { STANDARD = 1, VANILLA = 5,
  SCHEDULER = 7, MPI = 8, GRID = 9, JAVA = 10, PARALLEL =
  11, LOCALUNIVERSE = 12, VM = 13 };
  ```

- `command` The command to execute once the job has started.
- `arguments` The command-line arguments for `command`.
- `requirements` The requirements expression for the job. For further details and examples of the expression syntax, please refer to section 4.1.

Return Value If the function succeeds, the return value is `SUCCESS`; otherwise, see `StatusCode` for valid return values.

`discoverJobRequirements` Discover the requirements of a job, given a Class Ad. May be helpful in determining what should be sent along with the job. A prototype is

```
StatusAndRequirements discoverJobRequirements( ClassAd jobAd);
```

Parameters • `jobAd` The ClassAd of the job.

Return Value If the function succeeds, the return value is `SUCCESS`; otherwise, see `StatusCode` for valid return values. Additionally, on success, the return value contains the job's requirements.

Methods for File Transfer

`declareFile` Declare a file that may be used by a job. A prototype is

```
Status declareFile(Transaction transaction, int clusterId, int
jobId, String name, int size, HashType hashType, String hash);
```

Parameters • `transaction` The transaction in which this file is declared.

- `clusterId` The cluster identifier.
- `jobId` An identifier of the job that will use the file.
- `name` The name of the file.
- `size` The size of the file.
- `hashType` The type of hash mechanism used to verify file integrity, where `hashType` can be one of the following:
  `enum HashType { NOHASH, MD5HASH };`
- `hash` An optionally zero-length string encoding of the file hash.

Return Value If the function succeeds, the return value is `SUCCESS`; otherwise, see `StatusCode` for valid return values.

`sendFile` Send a file that a job may use. A prototype is

```
Status sendFile(Transaction transaction, int clusterId, int
jobId, String name, int offset, Base64 data);
```

Parameters • `transaction` The transaction in which this file is send.

- `clusterId` The cluster identifier.
- `jobId` An identifier of the job that will use the file.
- `name` The name of the file being sent.

- `offset` The starting offset within the file being sent.
- `length` The length from the offset to send.
- `data` The data block being sent. This could be the entire file or a sub-section of the file as defined by `offset` and `length`.

Return Value If the function succeeds, the return value is `SUCCESS`; otherwise, see `StatusCode` for valid return values.

`getFile` Get a file from a job's spool. A prototype is

```
StatusAndBase64 getFile(Transaction transaction, int
clusterId, int jobId, String name, int offset, int length);
```

Parameters
- `transaction` An optionally nullable transaction, meaning this call does not need to occur in a transaction.
- `clusterId` The cluster in which to search.
- `jobId` The job identifier the file is associated with.
- `name` The name of the file to retrieve.
- `offset` The starting offset withing the file being retrieved.
- `length` The length from the offset to retrieve.

Return Value If the function succeeds, the return value is `SUCCESS`; otherwise, see `StatusCode` for valid return values. Additionally, on success, the return value contains the file or a sub-section of the file as defined by `offset` and `length`.

`closeSpool` Close a job's spool. All the files in the job's spool can be deleted. A prototype is

```
Status closeSpool(Transaction transaction, int clusterId, int
jobId);
```

Parameters
- `transaction` An optionally nullable transaction, meaning this call does not need to occur in a transaction.
- `clusterId` The cluster identifier which the job is associated with.
- `jobId` The job identifier for which the spool is to be removed.

Return Value If the function succeeds, the return value is `SUCCESS`; otherwise, see `StatusCode` for valid return values.

`listSpool` List the files in a job's spool. A prototype is

```
StatusAndFileInfoArray listSpool(Transaction transaction, int
clusterId, int jobId);
```

Parameters
- `transaction` An optionally nullable transaction, meaning this call does not need to occur in a transaction.
- `clusterId` The cluster in which to search.
- `jobId` The job identifier to search for.

Return Value If the function succeeds, the return value is `SUCCESS`; otherwise, see `StatusCode` for valid return values. Additionally, on success, the return value contains a list of files and their respective sizes.

Methods for Job Management

newCluster Create a new job cluster. A prototype is

```
StatusAndInt newCluster(Transaction transaction);
```

Parameters • `transaction` The transaction in which this cluster is created.

Return Value If the function succeeds, the return value is `SUCCESS`; otherwise, see `StatusCode` for valid return values. Additionally, on success, the return value contains the cluster id.

removeCluster Remove a job cluster, and all the jobs within it. A prototype is

```
Status removeCluster(Transaction transaction, int clusterId,
String reason);
```

Parameters • `transaction` An optionally nullable transaction, meaning this call does not need to occur in a transaction.
• `clusterId` The cluster to remove.
• `reason` The reason for the removal.

Return Value If the function succeeds, the return value is `SUCCESS`; otherwise, see `StatusCode` for valid return values.

newJob Creates a new job within the most recently created job cluster. A prototype is

```
StatusAndInt newJob(Transaction transaction, int clusterId);
```

Parameters • `transaction` The transaction in which this job is created.
• `clusterId` The cluster identifier of the most recently created cluster.

Return Value If the function succeeds, the return value is `SUCCESS`; otherwise, see `StatusCode` for valid return values. Additionally, on success, the return value contains the job id.

removeJob Remove a job, regardless of the job's state. A prototype is

```
Status removeJob(Transaction transaction, int clusterId, int
jobId, String reason, boolean forceRemoval);
```

Parameters • `transaction` An optionally nullable transaction, meaning this call does not need to occur in a transaction.
• `clusterId` The cluster identifier to search in.
• `jobId` The job identifier to search for.
• `reason` The reason for the release.
• `forceRemoval` Set if the job should be forcibly removed.

Return Value If the function succeeds, the return value is `SUCCESS`; otherwise, see `StatusCode` for valid return values.

holdJob Put a job into the Hold state, regardless of the job's current state. A prototype is

```
Status holdJob(Transaction transaction, int clusterId, int
jobId, string reason, boolean emailUser, boolean emailAdmin,
boolean systemHold);
```

Parameters
- `transaction` An optionally nullable transaction, meaning this call does not need to occur in a transaction.
- `clusterId` The cluster in which to search.
- `jobId` The job identifier to search for.
- `reason` The reason for the release.
- `emailUser` Set if the submitting user should be notified.
- `emailAdmin` Set if the administrator should be notified.
- `systemHold` Set if the job should be put on hold.

Return Value If the function succeeds, the return value is `SUCCESS`; otherwise, see `StatusCode` for valid return values.

releaseJob Release a job that has been in the Hold state. A prototype is

```
Status releaseJob(Transaction transaction, int clusterId, int
jobId, String reason, boolean emailUser, boolean emailAdmin);
```

Parameters
- `transaction` An optionally nullable transaction, meaning this call does not need to occur in a transaction.
- `clusterId` The cluster in which to search.
- `jobId` The job identifier to search for.
- `reason` The reason for the release.
- `emailUser` Set if the submitting user should be notified.
- `emailAdmin` Set if the administrator should be notified.

Return Value If the function succeeds, the return value is `SUCCESS`; otherwise, see `StatusCode` for valid return values.

getJobAds A prototype is

```
StatusAndClassAdArray getJobAds(Transaction transaction,
String constraint);
```

Parameters
- `transaction` An optionally nullable transaction, meaning this call does not need to occur in a transaction.
- `constraint` A string constraining the number ClassAds to return. For further details and examples of the constraint syntax, please refer to section 4.1.

Return Value If the function succeeds, the return value is `SUCCESS`; otherwise, see `StatusCode` for valid return values. Additionally, on success, the return value contains all job ClassAds matching the given constraint.

getJobAd Finds a specific job ClassAd.

This method does much the same as the first element from the array returned by

```
getJobAds(transaction, "(ClusterId==clusterId && JobId==jobId)")
```

A prototype is

```
StatusAndClassAd getJobAd(Transaction transaction, int
clusterId, int jobId);
```

Parameters
- `transaction` An optionally nullable transaction, meaning this call does not need to occur in a transaction.
- `clusterId` The cluster in which to search.
- `jobId` The job identifier to search for.

Return Value If the function succeeds, the return value is `SUCCESS`; otherwise, see `StatusCode` for valid return values. Additionally, on success, the return value contains the requested ClassAd.

`requestReschedule` Request a *condor_reschedule* from the *condor_schedd* daemon. A prototype is

```
Status requestReschedule();
```

Return Value If the function succeeds, the return value is `SUCCESS`; otherwise, see `StatusCode` for valid return values.

Methods for ClassAd Management

`insertAd` A prototype is

```
Status insertAd(ClassAdType type, ClassAdStruct ad);
```

Parameters
- `type` The type of ClassAd to insert, where `type` can be one of the following:
```
enum ClassAdType { STARTD_AD_TYPE, QUILL_AD_TYPE,
SCHEDD_AD_TYPE, SUBMITTOR_AD_TYPE, LICENSE_AD_TYPE,
MASTER_AD_TYPE, CKPTSRVR_AD_TYPE, COLLECTOR_AD_TYPE,
STORAGE_AD_TYPE, NEGOTIATOR_AD_TYPE, HAD_AD_TYPE,
GENERIC_AD_TYPE };
```
- `ad` The ClassAd to insert.

Return Value If the function succeeds, the return value is `SUCCESS`; otherwise, see `StatusCode` for valid return values.

`queryStartdAds` A prototype is

```
ClassAdArray queryStartdAds(String constraint);
```

Parameters
- `constraint` A string constraining the number ClassAds to return. For further details and examples of the constraint syntax, please refer to section 4.1.

Return Value A list of all the *condor_startd* ClassAds matching the given constraint.

queryScheddAds A prototype is

```
ClassAdArray queryScheddAds(String constraint);
```

Parameters • `constraint` A string constraining the number ClassAds to return. For further details and examples of the constraint syntax, please refer to section 4.1.

Return Value A list of all the *condor_schedd* ClassAds matching the given constraint.

queryMasterAds A prototype is

```
ClassAdArray queryMasterAds(String constraint);
```

Parameters • `constraint` A string constraining the number ClassAds to return. For further details and examples of the constraint syntax, please refer to section 4.1.

Return Value A list of all the *condor_master* ClassAds matching the given constraint.

querySubmittorAds A prototype is

```
ClassAdArray querySubmittorAds(String constraint);
```

Parameters • `constraint` A string constraining the number ClassAds to return. For further details and examples of the constraint syntax, please refer to section 4.1.

Return Value A list of all the submitters ClassAds matching the given constraint.

queryLicenseAds A prototype is

```
ClassAdArray queryLicenseAds(String constraint);
```

Parameters • `constraint` A string constraining the number ClassAds to return.For further details and examples of the constraint syntax, please refer to section 4.1.

Return Value A list of all the license ClassAds matching the given constraint.

queryStorageAds A prototype is

```
ClassAdArray queryStorageAds(String constraint);
```

Parameters • `constraint` A string constraining the number ClassAds to return. For further details and examples of the constraint syntax, please refer to section 4.1.

Return Value A list of all the storage ClassAds matching the given constraint.

queryAnyAds A prototype is

```
ClassAdArray queryAnyAds(String constraint);
```

Parameters • `constraint` A string constraining the number ClassAds to return. For further details and examples of the constraint syntax, please refer to section 4.1.

Return Value A list of all the ClassAds matching the given constraint. to return.

Methods for Version Information

`getVersionString` A prototype is

```
StatusAndString getVersionString();
```

Return Value Returns the Condor version as a string.

`getPlatformString` A prototype is

```
StatusAndString getPlatformString();
```

Return Value Returns the platform information Condor is running on as string.

Common Data Structures

Many methods return a status. Table 4.5 lists and defines the `StatusCode` return values.

Value	Identifier	Definition
0	`SUCCESS`	All OK
1	`FAIL`	An error occurred that is not specific to another error code
2	`INVALIDTRANSACTION`	No such transaction exists
3	`UNKNOWNCLUSTER`	The specified cluster is not the currently active one
4	`UNKNOWNJOB`	The specified job does not exist within the specified cluster
5	`UNKNOWNFILE`	
6	`INCOMPLETE`	
7	`INVALIDOFFSET`	
8	`ALREADYEXISTS`	For this job, the specified file already exists

Table 4.5: `StatusCode` definitions

4.5.2 The DRMAA API

The following quote from the DRMAA Specification 1.0 abstract nicely describes the purpose of the API:

The Distributed Resource Management Application API (DRMAA), developed by a working group of the Global Grid Forum (GGF),

> provides a generalized API to distributed resource management systems (DRMSs) in order to facilitate integration of application programs. The scope of DRMAA is limited to job submission, job monitoring and control, and the retrieval of the finished

> job status. DRMAA provides application developers and distributed resource management builders with a programming model that enables the development of distributed applications tightly coupled to an underlying DRMS. For deployers of such distributed applications, DRMAA preserves flexibility and choice in system design.

The API allows users who write programs using DRMAA functions and link to a DRMAA library to submit, control, and retrieve information about jobs to a Grid system. The Condor implementation of a portion of the API allows programs (applications) to use the library functions provided to submit, monitor and control Condor jobs.

See the DRMAA site (http://www.drmaa.org) to find the API specification for DRMA 1.0 for further details on the API.

Implementation Details

The library was developed from the DRMA API Specification 1.0 of January 2004 and the DRMAA C Bindings v0.9 of September 2003. It is a static C library that expects a POSIX thread model on Unix systems and a Windows thread model on Windows systems. Unix systems that do not support POSIX threads are not guaranteed thread safety when calling the library's functions.

The object library file is called `libcondordrmaa.a`, and it is located within the `<release>/lib` directory in the Condor download. Its header file is called `lib_condor_drmaa.h`, and it is located within the `<release>/include` directory in the Condor download. Also within `<release>/include` is the file `lib_condor_drmaa.README`, which gives further details on the implementation.

Use of the library requires that a local *condor_schedd* daemon must be running, and the program linked to the library must have sufficient spool space. This space should be in `/tmp` or specified by the environment variables `TEMP`, `TMP`, or `SPOOL`. The program linked to the library and the local *condor_schedd* daemon must have read, write, and traverse rights to the spool space.

The library currently supports the following specification-defined job attributes:

DRMAA_REMOTE_COMMAND

DRMAA_JS_STATE

DRMAA_NATIVE_SPECIFICATION

DRMAA_BLOCK_EMAIL

DRMAA_INPUT_PATH

DRMAA_OUTPUT_PATH

DRMAA_ERROR_PATH

DRMAA_V_ARGV

DRMAA_V_ENV

DRMAA_V_EMAIL

The attribute `DRMAA_NATIVE_SPECIFICATION` can be used to direct all commands supported within submit description files. See the *condor_submit* manual page at section 9 for a complete list. Multiple commands can be specified if separated by newlines.

As in the normal submit file, arbitrary attributes can be added to the job's ClassAd by prefixing the attribute with +. In this case, you will need to put string values in quotation marks, the same as in a submit file.

Thus to tell Condor that the job will likely use 64 megabytes of memory (65536 kilobytes), to more highly rank machines with more memory, and to add the arbitrary attribute of department set to chemistry, you would set AttrDRMAA_NATIVE_SPECIFICATION to the C string:

```
drmaa_set_attribute(jobtemplate, DRMAA_NATIVE_SPECIFICATION,
    "image_size=65536\nrank=Memory\n+department=\"chemistry\"",
    err_buf, sizeof(err_buf)-1);
```

4.5.3 The Condor User and Job Log Reader API

Condor has the ability to log a Condor job's significant events during its lifetime. This is enabled in the job's submit description file with the **Log** command.

This section describes the API defined by the C++ `ReadUserLog` class, which provides a programming interface for applications to read and parse events, polling for events, and saving and restoring reader state.

Constants and Enumerated Types

The following define enumerated types useful to the API.

- `ULogEventOutcome` (defined in `condor_event.h`):
  - `ULOG_OK`: Event is valid
  - `ULOG_NO_EVENT`: No event occurred (like EOF)
  - `ULOG_RD_ERROR`: Error reading log file
  - `ULOG_MISSED_EVENT`: Missed event
  - `ULOG_UNK_ERROR`: Unknown Error
- `ReadUserLog::FileStatus`

- `LOG_STATUS_ERROR`: An error was encountered
- `LOG_STATUS_NOCHANGE`: No change in file size
- `LOG_STATUS_GROWN`: File has grown
- `LOG_STATUS_SHRUNK`: File has shrunk

Constructors and Destructors

All `ReadUserLog` constructors invoke one of the `initialize()` methods. Since C++ constructors cannot return errors, an application using any but the default constructor should call `isIinitialized()` to verify that the object initialized correctly, and for example, had permissions to open required files.

Note that because the constructors cannot return status information, most of these constructors will be eliminated in the future. All constructors, except for the default constructor with no parameters, will be removed. The application will need to call the appropriate `initialize()` method.

- `ReadUserLog::ReadUserLog(bool isEventLog)`
  Synopsis: Constructor default
  Returns: None
  Constructor parameters:
  - `bool isEventLog` (*Optional with default* = `false`)
    If `true`, the `ReadUserLog` object is initialized to read the schedd-wide event log.
    NOTE: If `isEventLog` is `true`, the initialization may silently fail, so the value of `ReadUserLog::isInitialized` should be checked to verify that the initialization was successful.
    NOTE: The `isEventLog` parameter will be removed in the future.
- `ReadUserLog::ReadUserLog(FILE *fp, bool is_xml, bool enable_close)`
  Synopsis: Constructor of a limited functionality reader: no rotation handling, no locking
  Returns: None
  Constructor parameters:
  - `FILE * fp`
    File pointer to the previously opened log file to read.
  - `bool is_xml`
    If `true`, the file is treated as XML; otherwise, it will be read as an old style file.
  - `bool enable_close` (*Optional with default* = `false`)
    If `true`, the reader will open the file read-only.

  NOTE: The `ReadUserLog::isInitialized` method should be invoked to verify that this constructor was initialized successfully.
  NOTE: This constructor will be removed in the future.

- `ReadUserLog::ReadUserLog(const char *filename, bool read_only)`
  Synopsis: Constructor to read a specific log file
  Returns: None
  Constructor parameters:
  - `const char * filename`
    Path to the log file to read
  - `bool read_only` (*Optional with default* = `false`)
    If `true`, the reader will open the file read-only and disable locking.

  NOTE: This constructor will be removed in the future.

- `ReadUserLog::ReadUserLog(const FileState &state, bool read_only)`
  Synopsis: Constructor to continue from a persisted reader state
  Returns: None
  Constructor parameters:
  - `const FileState & state`
    Reference to the persisted state to restore from
  - `bool read_only` (*Optional with default* = `false`)
    If `true`, the reader will open the file read-only and disable locking.

  NOTE: The `ReadUserLog::isInitialized` method should be invoked to verify that this constructor was initialized successfully.
  NOTE: This constructor will be removed in the future.

- `ReadUserLog::~ReadUserLog(void)`
  Synopsis: Destructor
  Returns: None
  Destructor parameters:
  - None.

Initializers

These methods are used to perform the initialization of the `ReadUserLog` objects. These initializers are used by all constructors that do real work. Applications should never use those constructors, should use the default constructor, and should instead use one of these initializer methods.

All of these functions will return `false` if there are problems such as being unable to open the log file, or `true` if successful.

- `bool ReadUserLog::initialize(void)`
  Synopsis: Initialize to read the EventLog file.
  NOTE: This method will likely be eliminated in the future, and this functionality will be

moved to a new `ReadEventLog` class.
Returns: `bool`; `true`: success, `false`: failed
Method parameters:

- None.

- `bool ReadUserLog::initialize(const char *filename, bool handle_rotation, bool check_for_rotated, bool read_only)`
Synopsis: Initialize to read a specific log file.
Returns: `bool`; `true`: success, `false`: failed
Method parameters:
  - `const char * filename`
  Path to the log file to read
  - `bool handle_rotation` (*Optional with default* = `false`)
  If `true`, enable the reader to handle rotating log files, which is only useful for global user logs
  - `bool check_for_rotated` (*Optional with default* = `false`)
  If `true`, try to open the rotated files (with file names appended with `.old` or `.1`, `.2`, . . .) first.
  - `bool read_only` (*Optional with default* = `false`)
  If `true`, the reader will open the file read-only and disable locking.

- `bool ReadUserLog::initialize(const char *filename, int max_rotation, bool check_for_rotated, bool read_only)`
Synopsis: Initialize to read a specific log file.
Returns: `bool`; `true`: success, `false`: failed
Method parameters:
  - `const char * filename`
  Path to the log file to read
  - `int max_rotation`
  Limits what previously rotated files will be considered by the number given in the file name suffix. A value of 0 disables looking for rotated files. A value of 1 limits the rotated file to be that with the file name suffix of `.old`. As only event logs are rotated, this parameter is only useful for event logs.
  - `bool check_for_rotated` (*Optional with default* = `false`)
  If `true`, try to open the rotated files (with file names appended with `.old` or `.1`, `.2`, . . .) first.
  - `bool read_only` (*Optional with default* = `false`)
  If `true`, the reader will open the file read-only and disable locking.

- `bool ReadUserLog::initialize(const FileState &state, bool read_only)`
Synopsis: Initialize to continue from a persisted reader state.
Returns: `bool`; `true`: success, `false`: failed
Method parameters:

  - `const FileState & state`
    Reference to the persisted state to restore from
  - `bool read_only` (*Optional with default* = `false`)
    If `true`, the reader will open the file read-only and disable locking.

- `bool ReadUserLog::initialize(const FileState &state, int max_rotation, bool read_only)`
  Synopsis: Initialize to continue from a persisted reader state and set the rotation parameters.
  Returns: `bool`; `true`: success, `false`: failed
  Method parameters:

  - `const FileState & state`
    Reference to the persisted state to restore from
  - `int max_rotation`
    Limits what previously rotated files will be considered by the number given in the file name suffix. A value of 0 disables looking for rotated files. A value of 1 limits the rotated file to be that with the file name suffix of `.old`. As only event logs are rotated, this parameter is only useful for event logs.
  - `bool read_only` (*Optional with default* = `false`)
    If `true`, the reader will open the file read-only and disable locking.

Primary Methods

- `ULogEventOutcome ReadUserLog::readEvent(ULogEvent *& event)`
  Synopsis: Read the next event from the log file.
  Returns: `ULogEventOutcome`; Outcome of the log read attempt. `ULogEventOutcome` is an enumerated type.
  Method parameters:

  - `ULogEvent *& event`
    Pointer to an `ULogEvent` that is allocated by this call to `ReadUserLog::readEvent`. If no event is allocated, this pointer is set to `NULL`. Otherwise the event needs to be `delete()ed` by the application.

- `bool ReadUserLog::synchronize(void)`
  Synopsis: Synchronize the log file if the last event read was an error. This safe guard function should be called if there is some error reading an event, but there are events after it in the file. It will skip over the bad event, meaning it will read up to and including the event separator, so that the rest of the events can be read.
  Returns: `bool`; `true`: success, `false`: failed
  Method parameters:

  - None.

Accessors

- `ReadUserLog::FileStatus ReadUserLog::CheckFileStatus(void)`
  Synopsis: Check the status of the file, and whether it has grown, shrunk, etc.
  Returns: `ReadUserLog::FileStatus`; the status of the log file, an enumerated type.
  Method parameters:
  - None.
- `ReadUserLog::FileStatus ReadUserLog::CheckFileStatus(bool &is_empty)`
  Synopsis: Check the status of the file, and whether it has grown, shrunk, etc.
  Returns: `ReadUserLog::FileStatus`; the status of the log file, an enumerated type.
  Method parameters:
  - `bool & is_empty`
    Set to `true` if the file is empty, `false` otherwise.

Methods for saving and restoring persistent reader state

The `ReadUserLog::FileState` structure is used to save and restore the state of the `ReadUserLog` state for persistence. The application should always use `InitFileState()` to initialize this structure.

All of these methods take a reference to a state buffer as their only parameter.

All of these methods return `true` upon success.

Save state to persistent storage

To save the state, do something like this:

```
ReadUserLog               reader;
ReadUserLog::FileState    statebuf;

status = ReadUserLog::InitFileState( statebuf );

status = reader.GetFileState( statebuf );
write( fd, statebuf.buf, statebuf.size );
...
status = reader.GetFileState( statebuf );
write( fd, statebuf.buf, statebuf.size );
...

status = UninitFileState( statebuf );
```

Restore state from persistent storage

To restore the state, do something like this:

```
ReadUserLog::FileState     statebuf;
status = ReadUserLog::InitFileState( statebuf );

read( fd, statebuf.buf, statebuf.size );

ReadUserLog                reader;
status = reader.initialize( statebuf );

status = UninitFileState( statebuf );
....
```

API Reference

- `static bool  ReadUserLog::InitFileState(ReadUserLog::FileState &state)`
  Synopsis: Initialize a file state buffer
  Returns: `bool`; `true` if successful, `false` otherwise
  Method parameters:
  - `ReadUserLog::FileState & state`
    The file state buffer to initialize.
- `static bool ReadUserLog::UninitFileState(ReadUserLog::FileState &state)`
  Synopsis: Clean up a file state buffer and free allocated memory
  Returns: `bool`; `true` if successful, `false` otherwise
  Method parameters:
  - `ReadUserLog::FileState & state`
    The file state buffer to un-initialize.
- `bool  ReadUserLog::GetFileState(ReadUserLog::FileState &state) const`
  Synopsis: Get the current state to persist it or save it off to disk
  Returns: `bool`; `true` if successful, `false` otherwise
  Method parameters:
  - `ReadUserLog::FileState & state`
    The file state buffer to read the state into.
- `bool    ReadUserLog::SetFileState(const ReadUserLog::FileState &state)`
  Synopsis: Use this method to set the current state, after restoring it.
  NOTE: The state buffer is *NOT* automatically updated; a call *MUST* be made to the `GetFileState()` method each time before persisting the buffer to disk, or however else

is chosen to persist its contents.
Returns: `bool`; `true` if successful, `false` otherwise
Method parameters:

- `const ReadUserLog::FileState & state`
  The file state buffer to restore from.

Access to the persistent state data

If the application needs access to the data elements in a persistent state, it should instantiate a `ReadUserLogStateAccess` object.

- Constructors / Destructors
  - `ReadUserLogStateAccess::ReadUserLogStateAccess(const ReadUserLog::FileState &state)`
    Synopsis: Constructor default
    Returns: None
    Constructor parameters:
    * `const ReadUserLog::FileState & state`
      Reference to the persistent state data to initialize from.
  - `ReadUserLogStateAccess::~ReadUserLogStateAccess(void)`
    Synopsis: Destructor
    Returns: None
    Destructor parameters:
    * None.
- Accessor Methods
  - `bool ReadUserLogFileState::isInitialized(void) const`
    Synopsis: Checks if the buffer initialized
    Returns: `bool`; `true` if successfully initialized, `false` otherwise
    Method parameters:
    * None.
  - `bool ReadUserLogFileState::isValid(void) const`
    Synopsis: Checks if the buffer is valid for use by `ReadUserLog::initialize()`
    Returns: `bool`; `true` if successful, `false` otherwise
    Method parameters:
    * None.
  - `bool ReadUserLogFileState::getFileOffset(unsigned long &pos) const`
    Synopsis: Get position within individual file.
    NOTE: Can return an error if the result is too large to be stored in a `long`.
    Returns: `bool`; `true` if successful, `false` otherwise
    Method parameters:

  * `unsigned long &pos`
    Byte position within the current log file

– `bool ReadUserLogFileState::getFileEventNum(unsigned long &num) const`
Synopsis: Get event number in individual file.
NOTE: Can return an error if the result is too large to be stored in a `long`.
Returns: `bool`; `true` if successful, `false` otherwise
Method parameters:

  * `unsigned long &num`
    Event number of the current event in the current log file

– `bool ReadUserLogFileState::getLogPosition(unsigned long &pos) const`
Synopsis: Position of the start of the current file in overall log.
NOTE: Can return an error if the result is too large to be stored in a `long`.
Returns: `bool`; `true` if successful, `false` otherwise
Method parameters:

  * `unsigned long &pos`
    Byte offset of the start of the current file in the overall logical log stream.

– `bool ReadUserLogFileState::getEventNumber(unsigned long &num) const`
Synopsis: Get the event number of the first event in the current file
NOTE: Can return an error if the result is too large to be stored in a `long`.
Returns: bool; `true` if successful, `false` otherwise
Method parameters:

  * `unsigned long &num`
    This is the absolute event number of the first event in the current file in the overall logical log stream.

– `bool ReadUserLogFileState::getUniqId(char *buf, int size) const`
Synopsis: Get the unique ID of the associated state file.
Returns: bool; `true` if successful, `false` otherwise
Method parameters:

  * `char *buf`
    Buffer to fill with the unique ID of the current file.
  * `int size`
    Size in bytes of `buf`.
    This is to prevent `ReadUserLogFileState::getUniqId` from writing past the end of `buf`.

– `bool ReadUserLogFileState::getSequenceNumber(int &seqno) const`
Synopsis: Get the sequence number of the associated state file.
Returns: `bool`; `true` if successful, `false` otherwise
Method parameters:

    * `int & seqno`
      Sequence number of the current file

- Comparison Methods

  – `bool ReadUserLogFileState::getFileOffsetDiff(const ReadUserLogStateAccess &other, unsigned long &pos) const`
  Synopsis: Get the position difference of two states given by `this` and `other`.
  NOTE: Can return an error if the result is too large to be stored in a `long`.
  Returns: `bool`; `true` if successful, `false` otherwise
  Method parameters:
    * `const ReadUserLogStateAccess & other`
      Reference to the state to compare to.
    * `long & diff`
      Difference in the positions

  – `bool ReadUserLogFileState::getFileEventNumDiff(const ReadUserLogStateAccess &other, long &diff) const`
  Synopsis: Get event number in individual file.
  NOTE: Can return an error if the result is too large to be stored in a `long`.
  Returns: bool; `true` if successful, `false` otherwise
  Method parameters:
    * `const ReadUserLogStateAccess & other`
      Reference to the state to compare to.
    * `long & diff`
      Event number of the current event in the current log file

  – `bool ReadUserLogFileState::getLogPosition(const ReadUserLogStateAccess &other, long &diff) const`
  Synopsis: Get the position difference of two states given by `this` and `other`.
  NOTE: Can return an error if the result is too large to be stored in a `long`.
  Returns: bool; `true` if successful, `false` otherwise
  Method parameters:
    * `const ReadUserLogStateAccess & other`
      Reference to the state to compare to.
    * `long & diff`
      Difference between the byte offset of the start of the current file in the overall logical log stream and that of `other`.

  – `bool ReadUserLogFileState::getEventNumber(const ReadUserLogStateAccess &other, long &diff) const`
  Synopsis: Get the difference between the event number of the first event in two state buffers (this - other).
  NOTE: Can return an error if the result is too large to be stored in a `long`.
  Returns: bool; `true` if successful, `false` otherwise
  Method parameters:
    * `const ReadUserLogStateAccess & other`
      Reference to the state to compare to.

* `long & diff`
  Difference between the absolute event number of the first event in the current file in the overall logical log stream and that of `other`.

Future persistence API

The `ReadUserLog::FileState` will likely be replaced with a new C++ `ReadUserLog::NewFileState`, or a similarly named class that will self initialize.

Additionally, the functionality of `ReadUserLogStateAccess` will be integrated into this class.

4.5.4 Chirp

This section has not yet been written

4.5.5 The Command Line Interface

This section has not yet been written

4.5.6 The Condor GAHP

This section has not yet been written

4.5.7 The Condor Perl Module

The Condor Perl module facilitates automatic submitting and monitoring of Condor jobs, along with automated administration of Condor. The most common use of this module is the monitoring of Condor jobs. The Condor Perl module can be used as a meta scheduler for the submission of Condor jobs.

The Condor Perl module provides several subroutines. Some of the subroutines are used as callbacks; an event triggers the execution of a specific subroutine. Other of the subroutines denote actions to be taken by Perl. Some of these subroutines take other subroutines as arguments.

Subroutines

`Submit(submit_description_file)` This subroutine takes the action of submitting a job to Condor. The argument is the name of a submit description file. The *condor_submit* program

should be in the path of the user. If the user wishes to monitor the job with condor they must specify a log file in the command file. The cluster submitted is returned. For more information see the *condor_submit* man page.

Vacate(machine) This subroutine takes the action of sending a *condor_vacate* command to the machine specified as an argument. The machine may be specified either by host name, or by *sinful string*. For more information see the *condor_vacate* man page.

Reschedule(machine) This subroutine takes the action of sending a *condor_reschedule* command to the machine specified as an argument. The machine may be specified either by host name, or by *sinful string*. For more information see the *condor_reschedule* man page.

Monitor(cluster) Takes the action of monitoring this cluster. It returns when all jobs in cluster terminate.

Wait() Takes the action of waiting until all monitor subroutines finish, and then exits the Perl script.

DebugOn() Takes the action of turning debug messages on. This may be useful when attempting to debug the Perl script.

DebugOff() Takes the action of turning debug messages off.

RegisterEvicted(sub) Register a subroutine (called `sub`) to be used as a callback when a job from a specified cluster is evicted. The subroutine will be called with two arguments: cluster and job. The cluster and job are the cluster number and process number of the job that was evicted.

RegisterEvictedWithCheckpoint(sub) Same as RegisterEvicted except that the handler is called when the evicted job was checkpointed.

RegisterEvictedWithoutCheckpoint(sub) Same as RegisterEvicted except that the handler is called when the evicted job was not checkpointed.

RegisterExit(sub) Register a termination handler that is called when a job exits. The termination handler will be called with two arguments: cluster and job. The cluster and job are the cluster and process numbers of the existing job.

RegisterExitSuccess(sub) Register a termination handler that is called when a job exits without errors. The termination handler will be called with two arguments: cluster and job The cluster and job are the cluster and process numbers of the existing job.

RegisterExitFailure(sub) Register a termination handler that is called when a job exits with errors. The termination handler will be called with three arguments: cluster, job and retval. The cluster and job are the cluster and process numbers of the existing job and the retval is the exit code of the job.

RegisterExitAbnormal(sub) Register an termination handler that is called when a job abnormally exits (segmentation fault, bus error, ...). The termination handler will be called with four arguments: cluster, job signal and core. The cluster and job are the cluster and process numbers of the existing job. The signal indicates the signal that the job died with and core indicates whether a core file was created and if so, what the full path to the core file is.

RegisterAbort(sub) Register a handler that is called when a job is aborted by a user.

RegisterJobErr(sub) Register a handler that is called when a job is not executable.

RegisterExecute(sub) Register an execution handler that is called whenever a job starts running on a given host. The handler is called with four arguments: cluster, job host, and sinful. Cluster and job are the cluster and process numbers for the job, host is the Internet address of the machine running the job, and sinful is the Internet address and command port of the *condor_starter* supervising the job.

RegisterSubmit(sub) Register a submit handler that is called whenever a job is submitted with the given cluster. The handler is called with cluster, job host, and sinful. Cluster and job are the cluster and process numbers for the job, host is the Internet address of the machine running the job, and sinful is the Internet address and command port of the *condor_schedd* responsible for the job.

Monitor(cluster) Begin monitoring this cluster. Returns when all jobs in cluster terminate.

Wait() Wait until all monitors finish and exit.

DebugOn() Turn debug messages on. This may be useful if you don't understand what your script is doing.

DebugOff() Turn debug messages off.

TestSubmit(command_file) This subroutine submits a job to Condor for testing, and places all variables from the command file into the Perl hash `%submit_info`. Does not reset the state of variables, so that testing preserves callbacks.

SubmitDagman(DAG_file, DAGMan_args) Takes the action of submitting a DAG using *condor_dagman*. The first argument is the name of the DAG input file, and the second argument is the command line arguments for *condor_dagman*. Information from the submit description file generated by *condor_dagman* is placed into the Perl hash `%submit_info` for access during callbacks.

TestSubmitDagman(DAG_file, DAGMan_args) This subroutine submits a *condor_dagman* to Condor for testing, and places information from the submit description file generated by *condor_dagman* into the Perl hash `%submit_info` for access during callbacks. The first argument is the name of the DAG input file, and the second argument is the command line arguments for *condor_dagman*. Does not reset the state of variables, so that testing preserves callbacks.

RegisterEvictedWithRequeue(sub) Register a subroutine (called `sub`) to be used as a callback when a job from a specified cluster is requeued. The subroutine will be called with two arguments: cluster and job. The cluster and job are the cluster number and process number of the job that was requeued.

RegisterShadow(sub) Register a subroutine (called `sub`) to be used as a callback when a shadow exception occurs.

RegisterHold(sub) Register a subroutine (called `sub`) to be used as a callback when a job enters the hold state.

RegisterRelease(sub) Register a subroutine (called `sub`) to be used as a callback when a job is released.

RegisterWantError(sub) Register a subroutine (called `sub`) to be used as a callback when a system call invoked using `runCommand` experiences an error.

runCommand(string) `string` identifies a syscall that is invoked. If the syscall exits abnormally or exits with an error, the callback registered with `RegisterWantError()` is called, and an error message is issued.

RegisterTimed(sub, seconds) Register a subroutine (called `sub`) to be called back at a delay of `seconds` time from this registration time. Only one callback may be registered, as subsequent calls modify the timer only.

RemoveTimed() Remove the single, timed callback registered with `RegisterTimed()`.

Examples

The following is an example that uses the Condor Perl module. The example uses the submit description file `mycmdfile.cmd` to specify the submission of a job. As the job is matched with a machine and begins to execute, a callback subroutine (called `execute`) sends a *condor_vacate* signal to the job, and it increments a counter which keeps track of the number of times this callback executes. A second callback keeps a count of the number of times that the job was evicted before the job completes. After the job completes, the termination callback (called `normal`) prints out a summary of what happened.

```
#!/usr/bin/perl
use Condor;

$CMD_FILE = 'mycmdfile.cmd';
$evicts = 0;
$vacates = 0;

# A subroutine that will be used as the normal execution callback
$normal = sub
{
    %parameters = @_;
    $cluster = $parameters{'cluster'};
    $job = $parameters{'job'};

    print "Job $cluster.$job exited normally without errors.\n";
    print "Job was vacated $vacates times and evicted $evicts times\n";
    exit(0);
};

$evicted = sub
{
    %parameters = @_;
```

```
    $cluster = $parameters{'cluster'};
    $job = $parameters{'job'};

    print "Job $cluster, $job was evicted.\n";
    $evicts++;
    &Condor::Reschedule();
};

$execute = sub
{
    %parameters = @_;
    $cluster = $parameters{'cluster'};
    $job = $parameters{'job'};
    $host = $parameters{'host'};
    $sinful = $parameters{'sinful'};

    print "Job running on $sinful, vacating...\n";
    &Condor::Vacate($sinful);
    $vacates++;
};

$cluster = Condor::Submit($CMD_FILE);
printf("Could not open. Access Denied\n");
break;
&Condor::RegisterExitSuccess($normal);
&Condor::RegisterEvicted($evicted);
&Condor::RegisterExecute($execute);
&Condor::Monitor($cluster);
&Condor::Wait();
```

This example program will submit the command file 'mycmdfile.cmd' and attempt to vacate any machine that the job runs on. The termination handler then prints out a summary of what has happened.

A second example Perl script facilitates the meta-scheduling of two of Condor jobs. It submits a second job if the first job successfully completes.

```
#!/s/std/bin/perl

# tell Perl where to find the Condor library
use lib '/unsup/condor/lib';
# tell Perl to use what it finds in the Condor library
use Condor;

$SUBMIT_FILE1 = 'Asubmit.cmd';
$SUBMIT_FILE2 = 'Bsubmit.cmd';

# Callback used when first job exits without errors.
$firstOK = sub
{
    %parameters = @_;
    $cluster = $parameters{'cluster'};
    $job = $parameters{'job'};

    $cluster = Condor::Submit($SUBMIT_FILE2);
    if (($cluster) == 0)
```

```
    {
        printf("Could not open $SUBMIT_FILE2.\n");
    }

    &Condor::RegisterExitSuccess($secondOK);
    &Condor::RegisterExitFailure($secondfails);
    &Condor::Monitor($cluster);
};

$firstfails = sub
{
    %parameters = @_;
    $cluster = $parameters{'cluster'};
    $job = $parameters{'job'};

    print "The first job, $cluster.$job failed, exiting with an error. \n";
    exit(0);
};

# Callback used when second job exits without errors.
$secondOK = sub
{
    %parameters = @_;
    $cluster = $parameters{'cluster'};
    $job = $parameters{'job'};

    print "The second job, $cluster.$job successfully completed. \n";
    exit(0);
};

# Callback used when second job exits WITH an error.
$secondfails = sub
{
    %parameters = @_;
    $cluster = $parameters{'cluster'};
    $job = $parameters{'job'};

    print "The second job ($cluster.$job) failed. \n";
    exit(0);
};


$cluster = Condor::Submit($SUBMIT_FILE1);
if (($cluster) == 0)
{
    printf("Could not open $SUBMIT_FILE1. \n");
}
&Condor::RegisterExitSuccess($firstOK);
&Condor::RegisterExitFailure($firstfails);


&Condor::Monitor($cluster);
&Condor::Wait();
```

Some notes are in order about this example. The same task could be accomplished using the Condor DAGMan metascheduler. The first job is the parent, and the second job is the child. The input file to DAGMan is significantly simpler than this Perl script.

A third example using the Condor Perl module expands upon the second example. Whereas the second example could have been more easily implemented using DAGMan, this third example shows the versatility of using Perl as a metascheduler.

In this example, the result generated from the successful completion of the first job are used to decide which subsequent job should be submitted. This is a very simple example of a branch and bound technique, to focus the search for a problem solution.

```
#!/s/std/bin/perl

# tell Perl where to find the Condor library
use lib '/unsup/condor/lib';
# tell Perl to use what it finds in the Condor library
use Condor;

$SUBMIT_FILE1 = 'Asubmit.cmd';
$SUBMIT_FILE2 = 'Bsubmit.cmd';
$SUBMIT_FILE3 = 'Csubmit.cmd';

# Callback used when first job exits without errors.
$firstOK = sub
{
    %parameters = @_;
    $cluster = $parameters{'cluster'};
    $job = $parameters{'job'};

    # open output file from first job, and read the result
    if ( -f "A.output" )
    {
        open(RESULTFILE, "A.output") or die "Could not open result file.";
        $result = <RESULTFILE>;
        close(RESULTFILE);
        # next job to submit is based on output from first job
        if ($result < 100)
        {
            $cluster = Condor::Submit($SUBMIT_FILE2);
            if (($cluster) == 0)
            {
                printf("Could not open $SUBMIT_FILE2.\n");
            }

            &Condor::RegisterExitSuccess($secondOK);
            &Condor::RegisterExitFailure($secondfails);
            &Condor::Monitor($cluster);
        }
        else
        {
            $cluster = Condor::Submit($SUBMIT_FILE3);
            if (($cluster) == 0)
            {
                printf("Could not open $SUBMIT_FILE3.\n");
            }

            &Condor::RegisterExitSuccess($thirdOK);
            &Condor::RegisterExitFailure($thirdfails);
            &Condor::Monitor($cluster);
```

```
        }
    }
    else
    {

        printf("Results file does not exist.\n");
    }
};

$firstfails = sub
{
    %parameters = @_;
    $cluster = $parameters{'cluster'};
    $job = $parameters{'job'};

    print "The first job, $cluster.$job failed, exiting with an error. \n";
    exit(0);
};


# Callback used when second job exits without errors.
$secondOK = sub
{
    %parameters = @_;
    $cluster = $parameters{'cluster'};
    $job = $parameters{'job'};

    print "The second job, $cluster.$job successfully completed. \n";
    exit(0);
};


# Callback used when third job exits without errors.
$thirdOK = sub
{
    %parameters = @_;
    $cluster = $parameters{'cluster'};
    $job = $parameters{'job'};

    print "The third job, $cluster.$job successfully completed. \n";
    exit(0);
};


# Callback used when second job exits WITH an error.
$secondfails = sub
{
    %parameters = @_;
    $cluster = $parameters{'cluster'};
    $job = $parameters{'job'};

    print "The second job ($cluster.$job) failed. \n";
    exit(0);
};

# Callback used when third job exits WITH an error.
$thirdfails = sub
{
```

```
    %parameters = @_;
    $cluster = $parameters{'cluster'};
    $job = $parameters{'job'};

    print "The third job ($cluster.$job) failed. \n";
    exit(0);
};


$cluster = Condor::Submit($SUBMIT_FILE1);
if (($cluster) == 0)
{
    printf("Could not open $SUBMIT_FILE1. \n");
}
&Condor::RegisterExitSuccess($firstOK);
&Condor::RegisterExitFailure($firstfails);


&Condor::Monitor($cluster);
&Condor::Wait();
```

CHAPTER

FIVE

Grid Computing

5.1 Introduction

A goal of grid computing is to allow the utilization of resources that span many administrative domains. A Condor pool often includes resources owned and controlled by many different people. Yet collaborating researchers from different organizations may not find it feasible to combine all of their computers into a single, large Condor pool. Condor shines in grid computing, continuing to evolve with the field.

Due to the field's rapid evolution, Condor has its own native mechanisms for grid computing as well as developing interactions with other grid systems.

Flocking is a native mechanism that allows Condor jobs submitted from within one pool to execute on another, separate Condor pool. Flocking is enabled by configuration within each of the pools. An advantage to flocking is that jobs migrate from one pool to another based on the availability of machines to execute jobs. When the local Condor pool is not able to run the job (due to a lack of currently available machines), the job flocks to another pool. A second advantage to using flocking is that the user (who submits the job) does not need to be concerned with any aspects of the job. The user's submit description file (and the job's **universe**) are independent of the flocking mechanism.

Other forms of grid computing are enabled by using the **grid universe** and further specified with the **grid_type**. For any Condor job, the job is submitted on a machine in the local Condor pool. The location where it is executed is identified as the remote machine or remote resource. These various grid computing mechanisms offered by Condor are distinguished by the software running on the remote resource.

When Condor is running on the remote resource, and the desired grid computing mechanism is

to move the job from the local pool's job queue to the remote pool's job queue, it is called Condor-C. The job is submitted using the **grid universe**, and the **grid_type** is **condor**. Condor-C jobs have the advantage that once the job has moved to the remote pool's job queue, a network partition does not affect the execution of the job. A further advantage of Condor-C jobs is that the **universe** of the job at the remote resource is not restricted.

When other middleware is running on the remote resource, such as Globus, Condor can still submit and manage jobs to be executed on remote resources. A **grid universe** job, with a **grid_type** of **gt2** or **gt4** calls on Globus software to execute the job on a remote resource. Like Condor-C jobs, a network partition does not affect the execution of the job. The remote resource must have Globus software running.

Condor also facilitates the temporary addition of a Globus-controlled resource to a local pool. This is called *glidein*. Globus software is utilized to execute Condor daemons on the remote resource. The remote resource appears to have joined the local Condor pool. A user submitting a job may then explicitly specify the remote resource as the execution site of a job.

Starting with Condor Version 6.7.0, the **grid** universe replaces the **globus** universe. Further specification of a **grid** universe job is done within the **grid_resource** command in a submit description file.

5.2 Connecting Condor Pools with Flocking

Flocking is Condor's way of allowing jobs that cannot immediately run (within the pool of machines where the job was submitted) to instead run on a different Condor pool. If a machine within Condor pool A can send jobs to be run on Condor pool B, then we say that jobs from machine A flock to pool B. Flocking can occur in a one way manner, such as jobs from machine A flocking to pool B, or it can be set up to flock in both directions. Configuration variables allow the *condor_schedd* daemon (which runs on each machine that may submit jobs) to implement flocking.

NOTE: Flocking to pools which use Condor's high availability mechanims is not adviced in current verions of Condor. See section 3.11.2 "High Availability of the Central Manager" of the Condor manual for a discussion of these problems.

5.2.1 Flocking Configuration

The simplest flocking configuration sets a few configuration variables. If jobs from machine A are to flock to pool B, then in machine A's configuration, set the following configuration variables:

FLOCK_TO is a comma separated list of the central manager machines of the pools that jobs from machine A may flock to.

FLOCK_COLLECTOR_HOSTS is the list of *condor_collector* daemons within the pools that jobs from machine A may flock to. In most cases, it is the same as FLOCK_TO, and it would be

defined with

```
FLOCK_COLLECTOR_HOSTS = $(FLOCK_TO)
```

FLOCK_NEGOTIATOR_HOSTS is the list of *condor_negotiator* daemons within the pools that jobs from machine A may flock to. In most cases, it is the same as FLOCK_TO, and it would be defined with

```
FLOCK_NEGOTIATOR_HOSTS = $(FLOCK_TO)
```

HOSTALLOW_NEGOTIATOR_SCHEDD provides a host-based access level and authorization list for the *condor_schedd* daemon to allow negotiation (for security reasons) with the machines within the pools that jobs from machine A may flock to. This configuration variable will not likely need to change from its default value as given in the sample configuration:

```
##  Now, with flocking we need to let the SCHEDD trust the other
##  negotiators we are flocking with as well.  You should normally
##  not have to change this either.
HOSTALLOW_NEGOTIATOR_SCHEDD = $(COLLECTOR_HOST), $(FLOCK_NEGOTIATOR_HOSTS)
```

This example configuration presumes that the *condor_collector* and *condor_negotiator* daemons are running on the same machine. See section 3.6.7 on page 336 for a discussion of security macros and their use.

The configuration macros that must be set in pool B are ones that authorize jobs from machine A to flock to pool B.

The host-based configuration macros are more easily set by introducing a list of machines where the jobs may flock from. FLOCK_FROM is a comma separated list of machines, and it is used in the default configuration setting of the security macros that do host-based authorization:

```
HOSTALLOW_WRITE_COLLECTOR = $(HOSTALLOW_WRITE), $(FLOCK_FROM)
HOSTALLOW_WRITE_STARTD    = $(HOSTALLOW_WRITE), $(FLOCK_FROM)
HOSTALLOW_READ_COLLECTOR  = $(HOSTALLOW_READ), $(FLOCK_FROM)
HOSTALLOW_READ_STARTD     = $(HOSTALLOW_READ), $(FLOCK_FROM)
```

Wild cards may be used when setting the FLOCK_FROM configuration variable. For example, `*.cs.wisc.edu` specifies all hosts from the `cs.wisc.edu` domain.

If the user-based configuration macros for security are used, then the default will be:

```
ALLOW_NEGOTIATOR =  $(COLLECTOR_HOST), $(FLOCK_NEGOTIATOR_HOSTS)
```

Further, if using Kerberos or GSI authentication, then the setting becomes:

```
ALLOW_NEGOTIATOR = condor@$(UID_DOMAIN)/$(COLLECTOR_HOST)
```

To enable flocking in both directions, consider each direction separately, following the guidelines given.

5.2.2 Job Considerations

A particular job will only flock to another pool when it cannot currently run in the current pool.

At one point, all jobs that utilized flocking were standard universe jobs. This is no longer the case. The submission of jobs under other universes must consider the location of input, output and error files. The common case will be that machines within separate pools do not have a shared file system. Therefore, when submitting jobs, the user will need to consider file transfer mechanisms. These mechanisms are discussed in section 2.5.4 on page 25.

5.3 The Grid Universe

5.3.1 Condor-C, The condor Grid Type

Condor-C allows jobs in one machine's job queue to be moved to another machine's job queue. These machines may be far removed from each other, providing powerful grid computation mechanisms, while requiring only Condor software and its configuration.

Condor-C is highly resistant to network disconnections and machine failures on both the submission and remote sides. An expected usage sets up Personal Condor on a laptop, submits some jobs that are sent to a Condor pool, waits until the jobs are staged on the pool, then turns off the laptop. When the laptop reconnects at a later time, any results can be pulled back.

Condor-C scales gracefully when compared with Condor's flocking mechanism. The machine upon which jobs are submitted maintains a single process and network connection to a remote machine, without regard to the number of jobs queued or running.

Condor-C Configuration

There are two aspects to configuration to enable the submission and execution of Condor-C jobs. These two aspects correspond to the endpoints of the communication: there is the machine from which jobs are submitted, and there is the remote machine upon which the jobs are placed in the queue (executed).

Configuration of a machine from which jobs are submitted requires a few extra configuration variables:

```
CONDOR_GAHP=$(SBIN)/condor_c-gahp
C_GAHP_LOG=/tmp/CGAHPLog.$(USERNAME)
C_GAHP_WORKER_THREAD_LOG=/tmp/CGAHPWorkerLog.$(USERNAME)
```

The acronym GAHP stands for Grid ASCII Helper Protocol. A GAHP server provides grid-related services for a variety of underlying middle-ware systems. The configuration variable

CONDOR_GAHP gives a full path to the GAHP server utilized by Condor-C. The configuration variable C_GAHP_LOG defines the location of the log that the Condor GAHP server writes. The log for the Condor GAHP is written as the user on whose behalf it is running; thus the C_GAHP_LOG configuration variable must point to a location the end user can write to.

A submit machine must also have a *condor_collector* daemon to which the *condor_schedd* daemon can submit a query. The query is for the location (IP address and port) of the intended remote machine's *condor_schedd* daemon. This facilitates communication between the two machines. This *condor_collector* does not need to be the same collector that the local *condor_schedd* daemon reports to.

The machine upon which jobs are executed must also be configured correctly. This machine must be running a *condor_schedd* daemon. Unless specified explicitly in a submit file, CONDOR_HOST must point to a *condor_collector* daemon that it can write to, and the machine upon which jobs are submitted can read from. This facilitates communication between the two machines.

An important aspect of configuration is the security configuration relating to authentication. Condor-C on the remote machine relies on an authentication protocol to know the identity of the user under which to run a job. The following is a working example of the security configuration for authentication. This authentication method, CLAIMTOBE, trusts the identity claimed by a host or IP address.

```
SEC_DEFAULT_NEGOTIATION = OPTIONAL
SEC_DEFAULT_AUTHENTICATION_METHODS = CLAIMTOBE
```

Condor-C Job Submission

Job submission of Condor-C jobs is the same as for any Condor job. The **universe** is **grid**. **grid_resource** specifies the remote *condor_schedd* daemon to which the job should be submitted, and its value consists of three fields. The first field is the grid type, which is **condor**. The second field is the name of the remote *condor_schedd* daemon. Its value is the same as the *condor_schedd* ClassAd attribute Name on the remote machine. The third field is the name of the remote pool's *condor_collector*.

The following represents a minimal submit description file for a job.

```
# minimal submit description file for a Condor-C job
universe = grid
executable = myjob
output = myoutput
error = myerror
log = mylog

grid_resource = condor joe@remotemachine.example.com remotecentralmanager.example.com
+remote_jobuniverse = 5
+remote_requirements = True
+remote_ShouldTransferFiles = "YES"
+remote_WhenToTransferOutput = "ON_EXIT"
queue
```

The remote machine needs to understand the attributes of the job. These are specified in the submit description file using the '+' syntax, followed by the string **remote_**. At a minimum, this will be the job's **universe** and the job's **requirements**. It is likely that other attributes specific to the job's **universe** (on the remote pool) will also be necessary. Note that attributes set with '+' are inserted directly into the job's ClassAd. Specify attributes as they must appear in the job's ClassAd, not the submit description file. For example, the **universe** is specified using an integer assigned for a job ClassAd `JobUniverse`. Similarly, place quotation marks around string expressions. As an example, a submit description file would ordinarily contain

```
when_to_transfer_output = ON_EXIT
```

This must appear in the Condor-C job submit description file as

```
+remote_WhenToTransferOutput = "ON_EXIT"
```

For convenience, the specific entries of **universe**, **remote_grid_resource**, **globus_rsl**, and **globus_xml** may be specified as **remote_** commands without the leading '+'. Instead of

```
+remote_universe = 5
```

the submit description file command may appear as

```
remote_universe = vanilla
```

Similarly, the command

```
+remote_gridresource = "condor schedd.example.com cm.example.com"
```

may be given as

```
remote_grid_resource = condor schedd.example.com cm.example.com
```

For the given example, the job is to be run as a **vanilla universe** job at the remote pool. The (remote pool's) *condor_schedd* daemon is likely to place its job queue data on a local disk and execute the job on another machine within the pool of machines. This implies that the file systems for the resulting submit machine (the machine specified by **remote_schedd**) and the execute machine (the machine that runs the job) will *not* be shared. Thus, the two inserted ClassAds

```
+remote_ShouldTransferFiles = "YES"
+remote_WhenToTransferOutput = "ON_EXIT"
```

are used to invoke Condor's file transfer mechanism.

As Condor-C is a recent addition to Condor, the universes, associated integer assignments, and notes about the existence of functionality are given in Table 5.1. The note "untested" implies that submissions under the given universe have not yet been throughly tested. They may already work.

Universe Name	Value	Notes
standard	1	untested
vanilla	5	works well
scheduler	7	works well
grid	9	
	grid_resource is condor	works well
	grid_resource is cream	untested
	grid_resource is gt2	works well
	grid_resource is gt4	untested
	grid_resource is gt5	untested
	grid_resource is nordugrid	untested
	grid_resource is unicore	untested
	grid_resource is lsf	works well
	grid_resource is pbs	works well
java	10	untested
parallel	11	untested
local	12	works well

Table 5.1: Functionality of remote job universes with Condor-C

For communication between *condor_schedd* daemons on the submit and remote machines, the location of the remote *condor_schedd* daemon is needed. This information resides in the *condor_collector* of the remote machine's pool. The third field of the **grid_resource** command in the submit description file says which *condor_collector* should be queried for the remote *condor_schedd* daemon's location. An example of this submit command is

```
grid_resource = condor schedd.example.com machine1.example.com
```

If the remote *condor_collector* is not listening on the standard port (9618), then the port it *is* listening on needs to be specified:

```
grid_resource = condor schedd.example.comd machine1.example.com:12345
```

File transfer of a job's executable, `stdin`, `stdout`, and `stderr` are automatic. When other files need to be transferred using Condor's file transfer mechanism (see section 2.5.4 on page 25), the mechanism is applied based on the resulting job universe on the remote machine.

Condor-C Jobs Between Differing Platforms

Condor-C jobs given to a remote machine running Windows must specify the Windows domain of the remote machine. This is accomplished by defining a ClassAd attribute for the job. Where the Windows domain is different at the submit machine from the remote machine, the submit description file defines the Windows domain of the remote machine with

```
+remote_NTDomain = "DomainAtRemoteMachine"
```

A Windows machine not part of a domain defines the Windows domain as the machine name.

Current Limitations in Condor-C

Submitting jobs to run under the grid universe has not yet been perfected. The following is a list of known limitations with Condor-C:

1. Authentication methods other than `CLAIMTOBE`, such as `GSI` and `KERBEROS`, are untested, and may not yet work.

5.3.2 Condor-G, the gt2, gt4, and gt5 Grid Types

Condor-G is the name given to Condor when **grid universe** jobs are sent to grid resources utilizing Globus software for job execution. The Globus Toolkit provides a framework for building grid systems and applications. See the Globus Alliance web page at http://www.globus.org for descriptions and details of the Globus software.

Condor provides the same job management capabilities for Condor-G jobs as for other jobs. From Condor, a user may effectively submit jobs, manage jobs, and have jobs execute on widely distributed machines.

It may appear that Condor-G is a simple replacement for the Globus Toolkit's *globusrun* command. However, Condor-G does much more. It allows the submission of many jobs at once, along with the monitoring of those jobs with a convenient interface. There is notification when jobs complete or fail and maintenance of Globus credentials that may expire while a job is running. On top of this, Condor-G is a fault-tolerant system; if a machine crashes, all of these functions are again available as the machine returns.

Globus Protocols and Terminology

The Globus software provides a well-defined set of protocols that allow authentication, data transfer, and remote job execution. Authentication is a mechanism by which an identity is verified. Given proper authentication, authorization to use a resource is required. Authorization is a policy that determines who is allowed to do what.

Condor (and Globus) utilize the following protocols and terminology. The protocols allow Condor to interact with grid machines toward the end result of executing jobs.

GSI The Globus Toolkit's Grid Security Infrastructure (GSI) provides essential building blocks for other grid protocols and Condor-G. This authentication and authorization system makes it

possible to authenticate a user just once, using public key infrastructure (PKI) mechanisms to verify a user-supplied grid credential. GSI then handles the mapping of the grid credential to the diverse local credentials and authentication/authorization mechanisms that apply at each site.

GRAM The Grid Resource Allocation and Management (GRAM) protocol supports remote submission of a computational request (for example, to run a program) to a remote computational resource, and it supports subsequent monitoring and control of the computation. GRAM is the Globus protocol that Condor-G uses to talk to remote Globus jobmanagers.

GASS The Globus Toolkit's Global Access to Secondary Storage (GASS) service provides mechanisms for transferring data to and from a remote HTTP, FTP, or GASS server. GASS is used by Condor for the **gt2** grid type to transfer a job's files to and from the machine where the job is submitted and the remote resource.

GridFTP GridFTP is an extension of FTP that provides strong security and high-performance options for large data transfers. It is used with the **gt4** grid type to transfer the job's files between the machine where the job is submitted and the remote resource.

RSL RSL (Resource Specification Language) is the language GRAM accepts to specify job information.

gatekeeper A gatekeeper is a software daemon executing on a remote machine on the grid. It is relevant only to the **gt2** grid type, and this daemon handles the initial communication between Condor and a remote resource.

jobmanager A jobmanager is the Globus service that is initiated at a remote resource to submit, keep track of, and manage grid I/O for jobs running on an underlying batch system. There is a specific jobmanager for each type of batch system supported by Globus (examples are Condor, LSF, and PBS).

Figure 5.1 shows how Condor interacts with Globus software towards running jobs. The diagram is specific to the **gt2** type of grid. Condor contains a GASS server, used to transfer the executable, `stdin`, `stdout`, and `stderr` to and from the remote job execution site. Condor uses the GRAM protocol to contact the remote gatekeeper and request that a new jobmanager be started. The GRAM protocol is also used to when monitoring the job's progress. Condor detects and intelligently handles cases such as if the remote resource crashes.

There are now three different versions of the GRAM protocol. Condor supports both the **gt2** and **gt4** protocols. It does not support **gt3**.

gt2 This initial GRAM protocol is used in Globus Toolkit versions 1 and 2. It is still used by many production systems. Where available in the other, more recent versions of the protocol, **gt2** is referred to as the pre-web services GRAM (or pre-WS GRAM).

gt3 **gt3** corresponds to Globus Toolkit version 3 as part of Globus' shift to web services-based protocols. It is replaced by the Globus Toolkit version 4. An installation of the Globus Toolkit version 3 (or OSGA GRAM) may also include the the pre-web services GRAM.

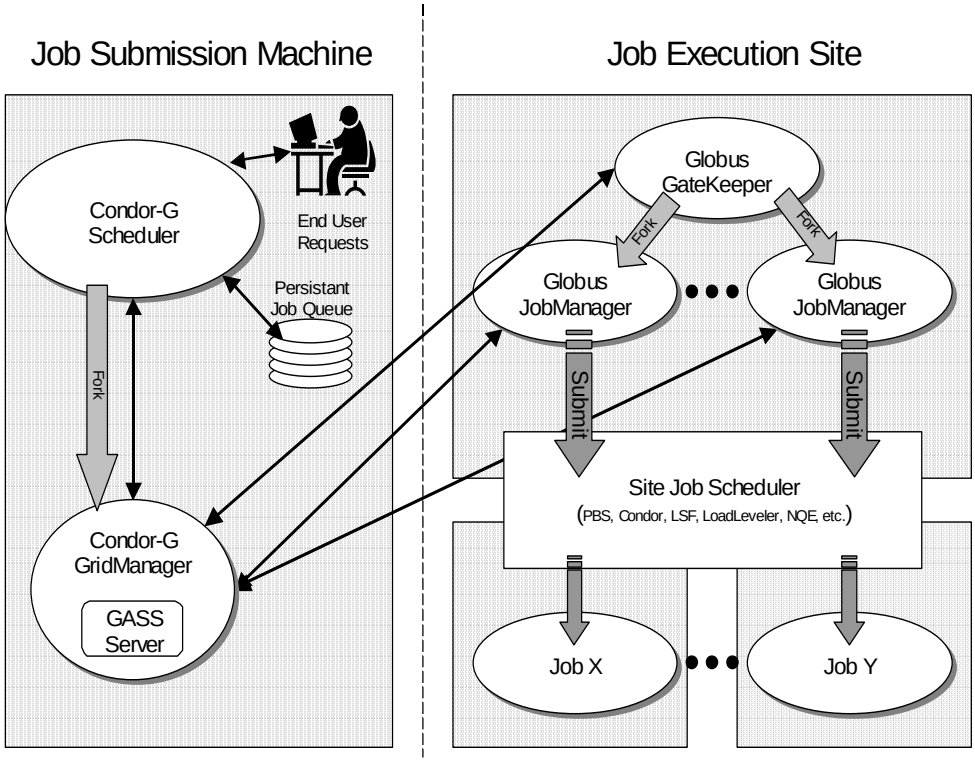

Figure 5.1: Condor-G interaction with Globus-managed resources

gt4 This GRAM protocol was introduced in Globus Toolkit version 4.0 as a more standards-compliant version of the GT3 web services-based GRAM. It is also called WS GRAM. A slightly different version of this GRAM protocol was introduced in Globus Toolkit 4.2 due to a change in the underlying Web Services standards. An installation of the Globus Toolkit version 4 may also include the the pre-web services GRAM.

gt5 This new GRAM5 protocol is still in alpha testing. It is an extension of the pre-WS GRAM protocol, and it attempts to fix lingering problems.

The gt2 Grid Type

Condor-G supports submitting jobs to remote resources running the Globus Toolkit versions 1 and 2, also called the pre-web services GRAM (or pre-WS GRAM). These Condor-G jobs are submitted the same as any other Condor job. The **universe** is **grid**, and the pre-web services GRAM protocol is specified by setting the type of grid as **gt2** in the **grid_resource** command.

Under Condor, successful job submission to the **grid universe** with **gt2** requires credentials. An X.509 certificate is used to create a proxy, and an account, authorization, or allocation to use a grid

resource is required. For general information on proxies and certificates, please consult the Globus page at

http://www-unix.globus.org/toolkit/docs/4.0/security/key-index.html

Before submitting a job to Condor under the **grid** universe, use *grid-proxy-init* to create a proxy.

Here is a simple submit description file. The example specifies a **gt2** job to be run on an NCSA machine.

```
executable = test
universe = grid
grid_resource = gt2 modi4.ncsa.uiuc.edu/jobmanager
output = test.out
log = test.log
queue
```

The **executable** for this example is transferred from the local machine to the remote machine. By default, Condor transfers the executable, as well as any files specified by an **input** command. Note that the executable must be compiled for its intended platform.

The command **grid_resource** is a required command for grid universe jobs. The second field specifies the scheduling software to be used on the remote resource. There is a specific jobmanager for each type of batch system supported by Globus. The full syntax for this command line appears as

```
grid_resource = gt2 machinename[:port]/jobmanagername[:X.509 distinguished name]
```

The portions of this syntax specification enclosed within square brackets ([and]) are optional. On a machine where the jobmanager is listening on a nonstandard port, include the port number. The `jobmanagername` is a site-specific string. The most common one is `jobmanager-fork`, but others are

```
jobmanager
jobmanager-condor
jobmanager-pbs
jobmanager-lsf
jobmanager-sge
```

The Globus software running on the remote resource uses this string to identify and select the correct service to perform. Other `jobmanagername` strings are used, where additional services are defined and implemented.

No input file is specified for this example job. Any output (file specified by an **output** command) or error (file specified by an **error** command) is transferred from the remote machine to the local machine as it is generated. This implies that these files may be incomplete in the case where the executable does not finish running on the remote resource. The ability to transfer standard output and standard error as they are produced may be disabled by adding to the submit description file:

```
stream_output = False
stream_error  = False
```

As a result, standard output and standard error will be transferred only after the job completes.

The job log file is maintained on the submit machine.

Example output from *condor_q* for this submission looks like:

```
% condor_q


-- Submitter: wireless48.cs.wisc.edu : <128.105.48.148:33012> : wireless48.cs.wi

 ID      OWNER         SUBMITTED     RUN_TIME ST PRI SIZE CMD
   7.0   smith        3/26 14:08   0+00:00:00 I  0   0.0  test

1 jobs; 1 idle, 0 running, 0 held
```

After a short time, the Globus resource accepts the job. Again running *condor_q* will now result in

```
% condor_q


-- Submitter: wireless48.cs.wisc.edu : <128.105.48.148:33012> : wireless48.cs.wi

 ID      OWNER         SUBMITTED     RUN_TIME ST PRI SIZE CMD
   7.0   smith        3/26 14:08   0+00:01:15 R  0   0.0  test

1 jobs; 0 idle, 1 running, 0 held
```

Then, very shortly after that, the queue will be empty again, because the job has finished:

```
% condor_q


-- Submitter: wireless48.cs.wisc.edu : <128.105.48.148:33012> : wireless48.cs.wi

 ID      OWNER            SUBMITTED     RUN_TIME ST PRI SIZE CMD

0 jobs; 0 idle, 0 running, 0 held
```

A second example of a submit description file runs the Unix *ls* program on a different Globus resource.

```
executable = /bin/ls
transfer_executable = false
universe = grid
grid_resource = gt2 vulture.cs.wisc.edu/jobmanager
output = ls-test.out
log = ls-test.log
queue
```

In this example, the executable (the binary) has been pre-staged. The executable is on the remote machine, and it is not to be transferred before execution. Note that the required **grid_resource** and **universe** commands are present. The command

```
transfer_executable = false
```

within the submit description file identifies the executable as being pre-staged. In this case, the **executable** command gives the path to the executable on the remote machine.

A third example submits a Perl script to be run as a submitted Condor job. The Perl script both lists and sets environment variables for a job. Save the following Perl script with the name `env-test.pl`, to be used as a Condor job executable.

```
#!/usr/bin/env perl

foreach $key (sort keys(%ENV))
{
   print "$key = $ENV{$key}\n"
}

exit 0;
```

Run the Unix command

```
chmod 755 env-test.pl
```

to make the Perl script executable.

Now create the following submit description file. Replace `example.cs.wisc.edu/jobmanager` with a resource you are authorized to use.

```
executable = env-test.pl
universe = grid
grid_resource = gt2 example.cs.wisc.edu/jobmanager
environment = foo=bar; zot=qux
output = env-test.out
log = env-test.log
queue
```

When the job has completed, the output file, `env-test.out`, should contain something like this:

```
GLOBUS_GRAM_JOB_CONTACT = https://example.cs.wisc.edu:36213/30905/1020633947/
GLOBUS_GRAM_MYJOB_CONTACT = URLx-nexus://example.cs.wisc.edu:36214
GLOBUS_LOCATION = /usr/local/globus
```

```
GLOBUS_REMOTE_IO_URL = /home/smith/.globus/.gass_cache/globus_gass_cache_1020633948
HOME = /home/smith
LANG = en_US
LOGNAME = smith
X509_USER_PROXY = /home/smith/.globus/.gass_cache/globus_gass_cache_1020633951
foo = bar
zot = qux
```

Of particular interest is the GLOBUS_REMOTE_IO_URL environment variable. Condor-G automatically starts up a GASS remote I/O server on the submit machine. Because of the potential for either side of the connection to fail, the URL for the server cannot be passed directly to the job. Instead, it is placed into a file, and the GLOBUS_REMOTE_IO_URL environment variable points to this file. Remote jobs can read this file and use the URL it contains to access the remote GASS server running inside Condor-G. If the location of the GASS server changes (for example, if Condor-G restarts), Condor-G will contact the Globus gatekeeper and update this file on the machine where the job is running. It is therefore important that all accesses to the remote GASS server check this file for the latest location.

The following example is a Perl script that uses the GASS server in Condor-G to copy input files to the execute machine. In this example, the remote job counts the number of lines in a file.

```
#!/usr/bin/env perl
use FileHandle;
use Cwd;

STDOUT->autoflush();
$gassUrl = `cat $ENV{GLOBUS_REMOTE_IO_URL}`;
chomp $gassUrl;

$ENV{LD_LIBRARY_PATH} = $ENV{GLOBUS_LOCATION}. "/lib";
$urlCopy = $ENV{GLOBUS_LOCATION}."/bin/globus-url-copy";

# globus-url-copy needs a full path name
$pwd = getcwd();
print "$urlCopy $gassUrl/etc/hosts file://$pwd/temporary.hosts\n\n";
`$urlCopy $gassUrl/etc/hosts file://$pwd/temporary.hosts`;

open(file, "temporary.hosts");
while(<file>) {
print $_;
}

exit 0;
```

The submit description file used to submit the Perl script as a Condor job appears as:

```
executable = gass-example.pl
universe = grid
grid_resource = gt2 example.cs.wisc.edu/jobmanager
output = gass.out
log = gass.log
queue
```

There are two optional submit description file commands of note: **x509userproxy** and **globus_rsl**. The **x509userproxy** command specifies the path to an X.509 proxy. The command is of the form:

```
x509userproxy = /path/to/proxy
```

If this optional command is not present in the submit description file, then Condor-G checks the value of the environment variable `X509_USER_PROXY` for the location of the proxy. If this environment variable is not present, then Condor-G looks for the proxy in the file `/tmp/x509up_uXXXX`, where the characters `XXXX` in this file name are replaced with the Unix user id.

The **globus_rsl** command is used to add additional attribute settings to a job's RSL string. The format of the **globus_rsl** command is

```
globus_rsl = (name=value)(name=value)
```

Here is an example of this command from a submit description file:

```
globus_rsl = (project=Test_Project)
```

This example's attribute name for the additional RSL is `project`, and the value assigned is `Test_Project`.

The gt4 Grid Type

Condor-G supports submitting jobs to remote resources running the Globus Toolkit version 4, which speak a protocol called WS GRAM. Please note that this Globus Toolkit version is *not* compatible with the Globus Toolkit version 3.0 or 3.2. Globus Toolkit versions 4.0 and 4.2 use slightly different (and incompatible) versions of WS GRAM, due to a change in the underlying Web Services standards. Condor is able to detect the difference and use the appropriate version for each remote resource automatically. See http://www-unix.globus.org/toolkit/docs/4.2/index.html for more information about the Globus Toolkit version 4.2.

For grid jobs destined for **gt4**, the submit description file is much the same as for **gt2** jobs. The **grid_resource** command is still required, and is given in the form of a URL. The syntax follows the form:

```
grid_resource = gt4 [https://]hostname[:port][/wsrf/services/ManagedJobFactoryService] scheduler-string
```

or

```
grid_resource = gt4 [https://]IPaddress[:port][/wsrf/services/ManagedJobFactoryService] scheduler-string
```

The portions of this syntax specification enclosed within square brackets ([and]) are optional.

The **scheduler-string** field of **grid_resource** indicates which job execution system should to be used on the remote system, to execute the job. One of these values is substituted for **scheduler-string**:

```
Fork
Condor
PBS
LSF
SGE
```

The **globus_xml** command can be used to add additional attributes to the XML-based RSL string that Condor writes to submit the job to GRAM. Here is an example of this command from a submit description file:

```
globus_xml = <project>Test_Project</project>
```

This example's attribute name for the additional RSL is `project`, and the value assigned is `Test_Project`.

File transfer occurs as expected for a Condor job (for the executable, `input`, and `output`), except that all output files other than `stdout` and `stderr` must be explicitly listed using **transfer_output_files**. The underlying transfer mechanism requires a *GridFTP* server to be running on the machine where the job is submitted. Condor will start one automatically. It will appear in the job queue as an additional job. It will leave the queue when there are no more **gt4** jobs in the queue. If the submit machine has a permanent *GridFTP* server running, instruct Condor to use it by setting the `GRIDFTP_URL_BASE` configuration variable. Here is an example setting:

```
GRIDFTP_URL_BASE = gsiftp://mycomp.foo.edu
```

On the submit machine, there is no requirement for any Globus Toolkit 4.0 components. Condor itself installs all necessary framework within the directory `$(LIB)/lib/gt4`. The machine where the job is submitted is required to have Java 1.5.0 or a higher version installed. You should not use the GNU Java interpreter (GCJ). The configuration variable `JAVA` must identify the location of the installation. See page 204 within section 3.3 for the complete description of the configuration variable `JAVA`.

The gt5 Grid Type

The Globus GRAM5 protocol works the same as the gt2 grid type. Its implementation differs from gt2 in the following 3 items:

- The Grid Monitor is disabled.

- Globus job managers are not stopped and restarted.
- The configuration variable `GRIDMANAGER_MAX_JOBMANAGERS_PER_RESOURCE` is not applied (for gt5 jobs).

Credential Management with *MyProxy*

Condor-G can use *MyProxy* software to automatically renew GSI proxies for **grid universe** jobs with grid type **gt2**. *MyProxy* is a software component developed at NCSA and used widely throughout the grid community. For more information see: http://myproxy.ncsa.uiuc.edu/

Difficulties with proxy expiration occur in two cases. The first case are long running jobs, which do not complete before the proxy expires. The second case occurs when great numbers of jobs are submitted. Some of the jobs may not yet be started or not yet completed before the proxy expires. One proposed solution to these difficulties is to generate longer-lived proxies. This, however, presents a greater security problem. Remember that a GSI proxy is sent to the remote Globus resource. If a proxy falls into the hands of a malicious user at the remote site, the malicious user can impersonate the proxy owner for the duration of the proxy's lifetime. The longer the proxy's lifetime, the more time a malicious user has to misuse the owner's credentials. To minimize the window of opportunity of a malicious user, it is recommended that proxies have a short lifetime (on the order of several hours).

The *MyProxy* software generates proxies using credentials (a user certificate or a long-lived proxy) located on a secure *MyProxy* server. Condor-G talks to the MyProxy server, renewing a proxy as it is about to expire. Another advantage that this presents is it relieves the user from having to store a GSI user certificate and private key on the machine where jobs are submitted. This may be particularly important if a shared Condor-G submit machine is used by several users.

In the a typical case, the following steps occur:

1. The user creates a long-lived credential on a secure *MyProxy* server, using the *myproxy-init* command. Each organization generally has their own *MyProxy* server.
2. The user creates a short-lived proxy on a local submit machine, using *grid-proxy-init* or *myproxy-get-delegation*.
3. The user submits a Condor-G job, specifying:

   MyProxy server name (host:port)

   MyProxy credential name (optional)

   MyProxy password
4. At the short-lived proxy expiration Condor-G talks to the *MyProxy* server to refresh the proxy.

Condor-G keeps track of the password to the *MyProxy* server for credential renewal. Although Condor-G tries to keep the password encrypted and secure, it is still possible (although highly unlikely) for the password to be intercepted from the Condor-G machine (more precisely, from the

machine that the *condor_schedd* daemon that manages the grid universe jobs runs on, which may be distinct from the machine from where jobs are submitted). The following safeguard practices are recommended.

1. Provide time limits for credentials on the *MyProxy* server. The default is one week, but you may want to make it shorter.
2. Create several different *MyProxy* credentials, maybe as many as one for each submitted job. Each credential has a unique name, which is identified with the `MyProxyCredentialName` command in the submit description file.
3. Use the following options when initializing the credential on the *MyProxy* server:

   ```
   myproxy-init -s <host> -x -r <cert subject> -k <cred name>
   ```

   The option **-x -r** <*cert subject*> essentially tells the *MyProxy* server to require two forms of authentication:

   (a) a password (initially set with *myproxy-init*)

   (b) an existing proxy (the proxy to be renewed)
4. A submit description file may include the password. An example contains commands of the form:

   ```
   executable      = /usr/bin/my-executable
   universe        = grid
   grid_resource   = gt4 condor-unsup-7
   MyProxyHost     = example.cs.wisc.edu:7512
   MyProxyServerDN = /O=doesciencegrid.org/OU=People/CN=Jane Doe 25900
   MyProxyPassword = password
   MyProxyCredentialName = my_executable_run
   queue
   ```

   Note that placing the password within the submit file is not really secure, as it relies upon whatever file system security there is. This may still be better than option 5.
5. Use the **-p** option to *condor_submit*. The submit command appears as

   ```
   condor_submit -p mypassword /home/user/myjob.submit
   ```

   The argument list for *condor_submit* defaults to being publicly available. An attacker with a log in to the local machine could generate a simple shell script to watch for the password.

Currently, Condor-G calls the *myproxy-get-delegation* command-line tool, passing it the necessary arguments. The location of the *myproxy-get-delegation* executable is determined by the configuration variable `MYPROXY_GET_DELEGATION` in the configuration file on the Condor-G machine. This variable is read by the *condor_gridmanager*. If *myproxy-get-delegation* is a dynamically-linked executable (verify this with `ldd myproxy-get-delegation`), point `MYPROXY_GET_DELEGATION` to a wrapper shell script that sets `LD_LIBRARY_PATH` to the correct *MyProxy* library or Globus library directory and then calls *myproxy-get-delegation*. Here is an example of such a wrapper script:

```
#!/bin/sh
export LD_LIBRARY_PATH=/opt/myglobus/lib
exec /opt/myglobus/bin/myproxy-get-delegation $@
```

The Grid Monitor

Condor's Grid Monitor is designed to improve the scalability of machines running Globus Toolkit 2 gatekeepers. Normally, this gatekeeper runs a jobmanager process for every job submitted to the gatekeeper. This includes both currently running jobs and jobs waiting in the queue. Each jobmanager runs a Perl script at frequent intervals (every 10 seconds) to poll the state of its job in the local batch system. For example, with 400 jobs submitted to a gatekeeper, there will be 400 jobmanagers running, each regularly starting a Perl script. When a large number of jobs have been submitted to a single gatekeeper, this frequent polling can heavily load the gatekeeper. When the gatekeeper is under heavy load, the system can become non-responsive, and a variety of problems can occur.

Condor's Grid Monitor temporarily replaces these jobmanagers. It is named the Grid Monitor, because it replaces the monitoring (polling) duties previously done by jobmanagers. When the Grid Monitor runs, Condor attempts to start a single process to poll all of a user's jobs at a given gatekeeper. While a job is waiting in the queue, but not yet running, Condor shuts down the associated jobmanager, and instead relies on the Grid Monitor to report changes in status. The jobmanager started to add the job to the remote batch system queue is shut down. The jobmanager restarts when the job begins running.

By default, standard output and standard error are streamed back to the submitting machine while the job is running. Streamed I/O requires the jobmanager. As a result, the Grid Monitor cannot replace the jobmanager for jobs that use streaming. If possible, disable streaming for all jobs; this is accomplished by placing the following lines in each job's submit description file:

```
stream_output = False
stream_error  = False
```

The Grid Monitor requires that the gatekeeper support the fork jobmanager with the name *jobmanager-fork*. If the gatekeeper does not support the fork jobmanager, the Grid Monitor will not be used for that site. The *condor_gridmanager* log file reports any problems using the Grid Monitor.

The Grid Monitor is enabled by default, and the configuration macro `GRID_MONITOR` identifies the location of the executable.

Limitations of Condor-G

Submitting jobs to run under the grid universe has not yet been perfected. The following is a list of known limitations:

1. No checkpoints.
2. No job exit codes. Job exit codes are not available when using **gt2**.
3. Limited platform availability. Windows support is not yet available.

5.3.3 The nordugrid Grid Type

NorduGrid is a project to develop free grid middleware named the Advanced Resource Connector (ARC). See the NorduGrid web page (http://www.nordugrid.org) for more information about NorduGrid software.

Condor jobs may be submitted to NorduGrid resources using the **grid** universe. The **grid_resource** command specifies the name of the NorduGrid resource as follows:

```
grid_resource = nordugrid ng.example.com
```

NorduGrid uses X.509 credentials for authentication, usually in the form a proxy certificate. For more information about proxies and certificates, please consult the Alliance PKI pages at http://archive.ncsa.uiuc.edu/SCD/Alliance/GridSecurity/. *condor_submit* looks in default locations for the proxy. The submit description file command **x509userproxy** is used to give the full path name to the directory containing the proxy, when the proxy is not in a default location. If this optional command is not present in the submit description file, then the value of the environment variable `X509_USER_PROXY` is checked for the location of the proxy. If this environment variable is not present, then the proxy in the file `/tmp/x509up_uXXXX` is used, where the characters `XXXX` in this file name are replaced with the Unix user id.

NorduGrid uses RSL syntax to describe jobs. The submit description file command **nordugrid_rsl** adds additional attributes to the job RSL that Condor constructs. The format this submit description file command is

```
nordugrid_rsl = (name=value)(name=value)
```

5.3.4 The unicore Grid Type

Unicore is a Java-based grid scheduling system. See http://unicore.sourceforge.net for more information about Unicore.

Condor jobs may be submitted to Unicore resources using the **grid** universe. The **grid_resource** command specifies the name of the Unicore resource as follows:

```
grid_resource = unicore usite.example.com vsite
```

usite.example.com is the host name of the Unicore gateway machine to which the Condor job is to be submitted. **vsite** is the name of the Unicore virtual resource to which the Condor job is to be submitted.

Unicore uses certificates stored in a Java keystore file for authentication. The following submit description file commands are required to properly use the keystore file.

keystore_file Specifies the complete path and file name of the Java keystore file to use.

keystore_alias A string that specifies which certificate in the Java keystore file to use.

keystore_passphrase_file Specifies the complete path and file name of the file containing the passphrase protecting the certificate in the Java keystore file.

5.3.5 The pbs Grid Type

The popular PBS (Portable Batch System) comes in several varieties: OpenPBS (http://www.openpbs.org), PBS Pro (http://www.altair.com/software/pbspro.htm), and Torque (http://www.clusterresources.com/pages/products/torque-resource-manager.php).

Condor jobs are submitted to a local PBS system using the **grid** universe and the **grid_resource** command by placing the following into the submit description file.

```
grid_resource = pbs
```

The pbs grid type requires two variables to be set in the Condor configuration file. `PBS_GAHP` is the path to the PBS GAHP server binary that is to be used to submit PBS jobs. `GLITE_LOCATION` is the path to the directory containing the GAHP's configuration file and auxillary binaries. In the Condor distribution, these files are located in `$(LIB)/glite`. The PBS GAHP's configuration file is in `$(GLITE_LOCATION)/etc/batch_gahp.config`. The PBS GAHP's auxillary binaries are to be in the directory `$(GLITE_LOCATION)/bin`. The Condor configuration file appears

```
GLITE_LOCATION = $(LIB)/glite
PBS_GAHP       = $(GLITE_LOCATION)/bin/batch_gahp
```

The PBS GAHP's configuration file contains two variables that must be modified to tell it where to find PBS on the local system. `pbs_binpath` is the directory that contains the PBS binaries. `pbs_spoolpath` is the PBS spool directory.

5.3.6 The lsf Grid Type

Condor jobs may be submitted to the Platform LSF batch system. See the Products page of the Platform web page at http://www.platform.com/Products/ for more information about Platform LSF.

Condor jobs are submitted to a local Platform LSF system using the **grid** universe and the **grid_resource** command by placing the following into the submit description file.

```
grid_resource = lsf
```

The lsf grid type requires two variables to be set in the Condor configuration file. `LSF_GAHP` is the path to the LSF GAHP server binary that is to be used to submit Platform LSF jobs. `GLITE_LOCATION` is the path to the directory containing the GAHP's configuration file and auxillary binaries. In the Condor distribution, these files are located in `$(LIB)/glite`. The LSF GAHP's configuration file is in `$(GLITE_LOCATION)/etc/batch_gahp.config`. The LSF GAHP's auxillary binaries are to be in the directory `$(GLITE_LOCATION)/bin`. The Condor configuration file appears

```
GLITE_LOCATION = $(LIB)/glite
LSF_GAHP       = $(GLITE_LOCATION)/bin/batch_gahp
```

The LSF GAHP's configuration file contains two variables that must be modified to tell it where to find LSF on the local system. `lsf_binpath` is the directory that contains the LSF binaries. `lsf_confpath` is the location of the LSF configuration file.

5.3.7 The amazon Grid Type

Condor jobs may be submitted to Amazon's Elastic Compute Cloud (EC2) service. EC2 is an on-line commercial service that allows the rental of computers by the hour to run computational applications. EC2 runs virtual machine images that have been uploaded to Amazon's online storage service (S3). See the Amazon EC2 web page at http://aws.amazon.com/ec2 for more information about EC2.

Amazon EC2 Job Submission

Condor jobs are submitted to EC2 using the **grid** universe and the **grid_resource** command by placing the following into the submit description file.

```
grid_resource = amazon https://ec2.amazonaws.com/
```

There many providers other than Amazon who support the EC2 interface. To use one of these others, substitute their EC2 URL for Amazon's in the **grid_resource** line.

Since the job is a virtual machine image, most of the submit description file commands specifying input or output files are not applicable. The **executable** command is still required, but its value is ignored. It can be used to identify different jobs in the output of *condor_q*.

The VM image for the job must already reside in Amazon's storage service (S3) and be registered with EC2. In the submit description file, provide the identifier for the image using the **amazon_ami_id** attribute. Also specify the files containing the X.509 certificate and private key used to authenticate with the EC2 service:

```
amazon_public_key = /path/to/x509/cert
amazon_private_key = /path/to/private/key
```

Condor and EC2 can create an ssh keypair to allow secure log in to the virtual machine once it is running. If the command **amazon_keypair_file** is set in the submit description file, Condor will write an ssh private key into the indicated file. The key can be used to log into the virtual machine. Note that modification will also be needed of the firewall rules for the job to all incoming ssh connections.

EC2 uses a firewall to restrict network access to the virtual machine instances it runs. By default, no incoming connections are allowed. One can define sets of firewall rules and give them names. EC2 calls these security groups. Then tell Condor what set of security groups should be applied to each VM using the **amazon_security_groups** submit description file command. If not provided, Condor uses the security group **default**.

EC2 offers several hardware configurations for instances to run on, with varying prices. Select which configuration to use with the **amazon_instance_type** submit description file command. Condor will use a default value of **m1.small** if not specified.

Each virtual machine instance can be given up to 16Kbytes of unique data, accessible by the instance connecting to a well-known address. This makes it easy for many instances to share the same VM image, but perform different work. This data can be specified to Condor in one of two ways. First, the data can be provided directly in the submit description file using the **amazon_user_data** command. Second, the data can be stored in a file, and the file name is specified with the **amazon_user_data_file** submit description file command. This second option allows the use of binary data. If both options are used, the two blocks of data are concatenated, with the data from **amazon_user_data** occurring first. Condor performs the base64 encoding that EC2 expects on the data.

Amazon EC2 Configuration Parameters

The amazon grid type requires several configuration variables to be set in the Condor configuration file:

```
AMAZON_GAHP=$(SBIN)/amazon_gahp
AMAZON_GAHP_LOG=/tmp/AmazonGahpLog.$(USERNAME)
```

If an HTTP proxy is needed to reach EC2, tell Condor to use it via the AMAZON_HTTP_PROXY configuration variable.

5.3.8 The cream Grid Type

CREAM is a job submission interface being developed at INFN for the gLite software stack. The CREAM homepage is http://grid.pd.infn.it/cream/. The protocol is based on web services.

The protocol requires an X.509 proxy for the job, so the submit description file command **x509userproxy** will be used.

A CREAM resource specification is of the form:

```
grid_resource = cream <web-services-address> <batch-system> <queue-name>
```

The `<web-services-address>` appears the same for most servers, differing only in the host name, as

```
<machinename[:port]>/ce-cream/services/CREAM2
```

Future versions of Condor may require only the host name, filling in other aspects of the web service for the user.

The `<batch-system>` is the name of the batch system that sits behind the CREAM server, into which it submits the jobs. Normal values are `pbs`, `lsf`, and `condor`.

The `<queue-name>` identifies which queue within the batch system should be used. Values for this will vary by site, with no typical values.

A full example for the specification of a CREAM **grid_resource** is

```
grid_resource = cream https://cream-12.pd.infn.it:8443/ce-cream/services/CREAM2
   pbs cream_1
```

This is a single line within the submit description file, although it is shown here on two lines for formatting reasons.

5.3.9 The deltacloud Grid Type

Condor jobs may be submitted to Deltacloud services. Deltacloud is a translation service for cloud services. Cloud services allow the rental of computers by the hour to run computation applications. Many cloud services define their own protocol for users to communicate with them. Deltacloud defines its own simple protocol and translates a user's commands into the appropriate protocol for the cloud service the user specifies. Anyone can set up a Deltacloud service and configure it to translate for a specific cloud service. See the Deltacloud web page at http://incubator.apache.org/deltacloud for more information about Deltacloud.

Deltacloud Job Submission

Condor jobs are submitted to Deltacloud using the **grid** universe and the **grid_resource** command into the submit description file following this example:

```
grid_resource = deltacloud https://deltacloud.foo.org/api
```

The URL in this example will be replaced with the URL of the Deltacloud service desired.

Since the job is a virtual machine image, most of the submit description file commands specifying input or output files are not applicable. The **executable** command is still required, but its value is ignored. It can be used to identify different jobs in the output of *condor_q*.

The VM image for the job must already be stored and registered with the cloud service. In the submit description file, provide the identifier for the image using the **deltacloud_image_id** command.

To authenticate with Deltacloud, Condor needs your credentials for the cloud service that the Deltacloud server is representing. The credentials are presented as a user name and the name of a file that holds a secret key. Both are specified in the submit description file:

```
deltacloud_username = your_username
deltacloud_password_file = /path/to/password/file
```

You can create and register an SSH key pair with the cloud service, which you can then use to securely log in to virtual machines, once running. The command **deltacloud_keyname** in the submit description file specifies the identifier of the SSH key pair to use.

The cloud service may have multiple locations where the virtual machine can run. The submit description file command **deltacloud_realm_id** selects one. If not specified, the service will select a sensible default.

The cloud service may offer several hardware configurations for instances to run on. Select which configuration to use with the **deltacloud_hardware_profile** submit description file command. If not specified, the cloud service will select a sensible default. The optional commands **deltacloud_hardware_profile_memory**, **deltacloud_hardware_profile_cpu**, and **deltacloud_hardware_profile_storage** customize the selected hardware profile.

Each virtual machine instance can be given some unique data, accessible by the instance connecting to a well-known address. This makes it easy for many instances to share the same VM image, but perform different work. This data can be specified with the submit description file command **deltacloud_user_data**. The amount of data that can be provided depends on the cloud service. EC2 services allow up to 16Kb of data.

Configuration for Deltacloud

The deltacloud grid type requires one configuration variable to be set, to specify the path and executable of the *deltacloud_gahp*:

```
DELTACLOUD_GAHP=$(SBIN)/deltacloud_gahp
```

5.3.10 Matchmaking in the Grid Universe

In a simple usage, the grid universe allows users to specify a single grid site as a destination for jobs. This is sufficient when a user knows exactly which grid site they wish to use, or a higher-level resource broker (such as the European Data Grid's resource broker) has decided which grid site should be used.

When a user has a variety of grid sites to choose from, Condor allows matchmaking of grid universe jobs to decide which grid resource a job should run on. Please note that this form of matchmaking is relatively new. There are some rough edges as continual improvement occurs.

To facilitate Condor's matching of jobs with grid resources, both the jobs and the grid resources are involved. The job's submit description file provides all commands needed to make the job work on a matched grid resource. The grid resource identifies itself to Condor by advertising a ClassAd. This ClassAd specifies all necessary attributes, such that Condor can properly make matches. The grid resource identification is accomplished by using *condor_advertise* to send a ClassAd representing the grid resource, which is then used by Condor to make matches.

Job Submission

To submit a grid universe job intended for a single, specific **gt2** resource, the submit description file for the job explicitly specifies the resource:

```
grid_resource = gt2 grid.example.com/jobmanager-pbs
```

If there were multiple **gt2** resources that might be matched to the job, the submit description file changes:

```
grid_resource   = $$(resource_name)
requirements    = TARGET.resource_name =!= UNDEFINED
```

The **grid_resource** command uses a substitution macro. The substitution macro defines the value of `resource_name` using attributes as specified by the matched grid resource. The **requirements** command further restricts that the job may only run on a machine (grid resource) that defines `grid_resource`. Note that this attribute name is invented for this example. To make

matchmaking work in this way, both the job (as used here within the submit description file) and the grid resource (in its created and advertised ClassAd) must agree upon the name of the attribute.

As a more complex example, consider a job that wants to run not only on a **gt2** resource, but on one that has the Bamboozle software installed. The complete submit description file might appear:

```
universe        = grid
executable      = analyze_bamboozle_data
output          = aaa.$(Cluster).out
error           = aaa.$(Cluster).err
log             = aaa.log
grid_resource   = $$(resource_name)
requirements    = (TARGET.HaveBamboozle == True) && (TARGET.resource_name =!= UNDEFINED)
queue
```

Any grid resource which has the `HaveBamboozle` attribute defined as well as set to `True` is further checked to have the `resource_name` attribute defined. Where this occurs, a match may be made (from the job's point of view). A grid resource that has one of these attributes defined, but not the other results in no match being made.

Note that the entire value of **grid_resource** comes from the grid resource's ad. This means that the job can be matched with a resource of any type, not just **gt2**.

Advertising Grid Resources to Condor

Any grid resource that wishes to be matched by Condor with a job must advertise itself to Condor using a ClassAd. To properly advertise, a ClassAd is sent periodically to the *condor_collector* daemon. A ClassAd is a list of pairs, where each pair consists of an attribute name and value that describes an entity. There are two entities relevant to Condor: a job, and a machine. A grid resource is a machine. The ClassAd describes the grid resource, as well as identifying the capabilities of the grid resource. It may also state both requirements and preferences (called **rank**) for the jobs it will run. See Section 2.3 for an overview of the interaction between matchmaking and ClassAds. A list of common machine ClassAd attributes is given in the Appendix on page 913.

To advertise a grid site, place the attributes in a file. Here is a sample ClassAd that describes a grid resource that is capable of running a **gt2** job.

```
# example grid resource ClassAd for a gt2 job
MyType          = "Machine"
TargetType      = "Job"
Name            = "Example1_Gatekeeper"
Machine         = "Example1_Gatekeeper"
resource_name   = "gt2 grid.example.com/jobmanager-pbs"
UpdateSequenceNumber  = 4
Requirements    = (TARGET.JobUniverse == 9)
Rank            = 0.000000
CurrentRank     = 0.000000
```

Some attributes are defined as expressions, while others are integers, floating point values, or strings. The type is important, and must be correct for the ClassAd to be effective. The attributes

```
MyType          = "Machine"
TargetType      = "Job"
```

identify the grid resource as a machine, and that the machine is to be matched with a job. In Condor, machines are matched with jobs, and jobs are matched with machines. These attributes are strings. Strings are surrounded by double quote marks.

The attributes `Name` and `Machine` are likely to be defined to be the same string value as in the example:

```
Name            = "Example1_Gatekeeper"
Machine         = "Example1_Gatekeeper"
```

Both give the fully qualified host name for the resource. The `Name` may be different on an SMP machine, where the individual CPUs are given names that can be distinguished from each other. Each separate grid resource must have a unique name.

Where the job depends on the resource to specify the value of the **grid_resource** command by the use of the substitution macro, the ClassAd for the grid resource (machine) defines this value. The example given as

```
grid_resource = "gt2 grid.example.com/jobmanager-pbs"
```

defines this value. Note that the invented name of this variable must match the one utilized within the submit description file. To make the matchmaking work, both the job (as used within the submit description file) and the grid resource (in this created and advertised ClassAd) must agree upon the name of the attribute.

A machine's ClassAd information can be time sensitive, and may change over time. Therefore, ClassAds expire and are thrown away. In addition, the communication method by which ClassAds are sent implies that entire ads may be lost without notice or may arrive out of order. Out of order arrival leads to the definition of an attribute which provides an ordering. This positive integer value is given in the example ClassAd as

```
UpdateSequenceNumber  = 4
```

This value must increase for each subsequent ClassAd. If state information for the ClassAd is kept in a file, a script executed each time the ClassAd is to be sent may use a counter for this value. An alternative for a stateless implementation sends the current time in seconds (since the epoch, as given by the C `time()` function call).

The requirements that the grid resource sets for any job that it will accept are given as

```
Requirements     = (TARGET.JobUniverse == 9)
```

This set of requirements state that any job is required to be for the **grid** universe.

The attributes

```
Rank             = 0.000000
CurrentRank      = 0.000000
```

are both necessary for Condor's negotiation to proceed, but are not relevant to grid matchmaking. Set both to the floating point value 0.0.

The example machine ClassAd becomes more complex for the case where the grid resource allows matches with more than one job:

```
# example grid resource ClassAd for a gt2 job
MyType         = "Machine"
TargetType     = "Job"
Name           = "Example1_Gatekeeper"
Machine        = "Example1_Gatekeeper"
resource_name  = "gt2 grid.example.com/jobmanager-pbs"
UpdateSequenceNumber  = 4
Requirements   = (CurMatches < 10) && (TARGET.JobUniverse == 9)
Rank           = 0.000000
CurrentRank    = 0.000000
WantAdRevaluate = True
CurMatches     = 1
```

In this example, the two attributes `WantAdRevaluate` and `CurMatches` appear, and the `Requirements` expression has changed.

`WantAdRevaluate` is a boolean value, and may be set to either `True` or `False`. When `True` in the ClassAd and a match is made (of a job to the grid resource), the machine (grid resource) is not removed from the set of machines to be considered for further matches. This implements the ability for a single grid resource to be matched to more than one job at a time. Note that the spelling of this attribute is incorrect, and remains incorrect to maintain backward compatibility.

To limit the number of matches made to the single grid resource, the resource must have the ability to keep track of the number of Condor jobs it has. This integer value is given as the `CurMatches` attribute in the advertised ClassAd. It is then compared in order to limit the number of jobs matched with the grid resource.

```
Requirements   = (CurMatches < 10) && (TARGET.JobUniverse == 9)
CurMatches     = 1
```

This example assumes that the grid resource already has one job, and is willing to accept a maximum of 9 jobs. If `CurMatches` does not appear in the ClassAd, Condor uses a default value of 0.

For multiple matching of a site ClassAd to work correctly, it is also necessary to add the following to the configuration file read by the *condor_negotiator*:

```
NEGOTIATOR_MATCHLIST_CACHING = False
NEGOTIATOR_IGNORE_USER_PRIORITIES = True
```

This ClassAd (likely in a file) is to be periodically sent to the *condor_collector* daemon using *condor_advertise*. A recommended implementation uses a script to create or modify the ClassAd together with *cron* to send the ClassAd every five minutes. The *condor_advertise* program must be installed on the machine sending the ClassAd, but the remainder of Condor does not need to be installed. The required argument for the *condor_advertise* command is *UPDATE_STARTD_AD*.

condor_advertise uses UDP to transmit the ClassAd. Where this is insufficient, specify the **-tcp** option to *condor_advertise* to use TCP for communication.

Advanced usage

What if a job fails to run at a grid site due to an error? It will be returned to the queue, and Condor will attempt to match it and re-run it at another site. Condor isn't very clever about avoiding sites that may be bad, but you can give it some assistance. Let's say that you want to avoid running at the last grid site you ran at. You could add this to your job description:

```
match_list_length = 1
Rank              = TARGET.Name != LastMatchName0
```

This will prefer to run at a grid site that was not just tried, but it will allow the job to be run there if there is no other option.

When you specify **match_list_length**, you provide an integer N, and Condor will keep track of the last N matches. The oldest match will be LastMatchName0, and next oldest will be LastMatchName1, and so on. (See the *condor_submit* manual page for more details.) The Rank expression allows you to specify a numerical ranking for different matches. When combined with **match_list_length**, you can prefer to avoid sites that you have already run at.

In addition, *condor_submit* has two options to help control grid universe job resubmissions and rematching. See the definitions of the submit description file commands **globus_resubmit** and **globus_rematch** at page 843 and page 843. These options are independent of **match_list_length**.

There are some new attributes that will be added to the Job ClassAd, and may be useful to you when you write your rank, requirements, globus_resubmit or globus_rematch option. Please refer to the Appendix on page 902 to see a list containing the following attributes:

- NumJobMatches
- NumGlobusSubmits
- NumSystemHolds
- HoldReason
- ReleaseReason
- EnteredCurrentStatus

- LastMatchTime
- LastRejMatchTime
- LastRejMatchReason

The following example of a command within the submit description file releases jobs 5 minutes after being held, increasing the time between releases by 5 minutes each time. It will continue to retry up to 4 times per Globus submission, plus 4. The plus 4 is necessary in case the job goes on hold before being submitted to Globus, although this is unlikely.

```
periodic_release = ( NumSystemHolds <= ((NumGlobusSubmits * 4) + 4) ) \
   && (NumGlobusSubmits < 4) && \
   ( HoldReason != "via condor_hold (by user $ENV(USER))" ) && \
   ((CurrentTime - EnteredCurrentStatus) > ( NumSystemHolds *60*5 ))
```

The following example forces Globus resubmission after a job has been held 4 times per Globus submission.

```
globus_resubmit = NumSystemHolds == (NumGlobusSubmits + 1) * 4
```

If you are concerned about unknown or malicious grid sites reporting to your *condor_collector*, you should use Condor's security options, documented in Section 3.6.

5.4 Glidein

Glidein is a mechanism by which one or more grid resources (remote machines) temporarily join a local Condor pool. The program *condor_glidein* is used to add a machine to a Condor pool. During the period of time when the added resource is part of the local pool, the resource is visible to users of the pool. But, by default, the resource is only available for use by the user that added the resource to the pool.

After glidein, the user may submit jobs for execution on the added resource the same way that all Condor jobs are submitted. To force a submitted job to run on the added resource, the submit description file could contain a requirement that the job run specifically on the added resource.

5.4.1 What *condor_glidein* Does

condor_glidein works by installing and executing necessary Condor daemons and configuration on the remote resource, such that the resource reports to and joins the local pool. *condor_glidein* accomplishes two separate tasks towards having a remote grid resource join the local Condor pool. They are the set up task and the execution task.

The set up task generates necessary configuration files and locates proper platform-dependent binaries for the Condor daemons. A script is also generated that can be used during the execution task to invoke the proper Condor daemons. These files are copied to the remote resource as necessary. The configuration variable GLIDEIN_SERVER_URLS defines a list of locations from which the necessary binaries are obtained. Default values cause binaries to be downloaded from the UW site. See section 3.3.25 on page 244 for a full definition of this configuration variable.

When the files are correctly in place, the execution task starts the Condor daemons. *condor_glidein* does this by submitting a Condor job to run under the grid universe. The job runs the *condor_master* on the remote grid resource. The *condor_master* invokes other daemons, which contact the local pool's *condor_collector* to join the pool. The Condor daemons exit gracefully when no jobs run on the daemons for a preset period of time.

Here is an example of how a glidein resource appears, similar to how any other machine appears. The name has a slightly different form, in order to handle the possibility of multiple instances of glidein daemons inhabiting a multi-processor machine.

```
% condor_status | grep denal
7591386@denal LINUX       INTEL  Unclaimed  Idle       3.700  24064  0+00:06:35
```

5.4.2 Configuration Requirements in the Local Pool

As remote grid resources join the local pool, these resources must report to the local pool's *condor_collector* daemon. Security demands that the local pool's *condor_collector* list all hosts from which they will accept communication. Therefore, all remote grid resources accepted for glidein must be given HOSTALLOW_WRITE permission. An expected way to do this is to modify the empty variable (within the sample configuration file) GLIDEIN_SITES to list all remote grid resources accepted for glidein. The list is a space or comma separated list of hosts. This list is then given the proper permissions by an additional redefinition of the HOSTALLOW_WRITE configuration variable, to also include the list of hosts as in the following example.

```
GLIDEIN_SITES = A.example.com, B.example.com, C.example.com
HOSTALLOW_WRITE = $(HOSTALLOW_WRITE) $(GLIDEIN_SITES)
```

Recall that for configuration file changes to take effect, *condor_reconfig* must be run.

If this configuration change to the security settings on the local Condor pool cannot be made, an additional Condor pool that utilizes personal Condor may be defined. The single machine pool may coexist with other instances of Condor. *condor_glidein* is executed to have the remote grid resources join this personal Condor pool.

5.4.3 Running Jobs on the Remote Grid Resource After Glidein

Once the Globus resource has been added to the local Condor pool with *condor_glidein*, job(s) may be submitted. To force a job to run on the Globus resource, specify that Globus resource as a machine requirement in the submit description file. Here is an example from within the submit description file that forces submission to the Globus resource denali.mcs.anl.gov:

```
requirements = ( machine == "denali.mcs.anl.gov" ) \
   && FileSystemDomain != "" \
   && Arch != "" && OpSys != ""
```

This example requires that the job run only on denali.mcs.anl.gov, and it prevents Condor from inserting the file system domain, architecture, and operating system attributes as requirements in the matchmaking process. Condor must be told not to use the submission machine's attributes in those cases where the Globus resource's attributes do not match the submission machine's attributes and your job really is capable of running on the target machine. You may want to use Condor's file-transfer capabilities in order to copy input and output files back and forth between the submission and execution machine.

5.5 Dynamic Deployment

See section 3.2.10 for a complete description of Condor's dynamic deployment tools.

Condor's dynamic deployment tools (*condor_cold_start* and *condor_glidein*) allow new pools of resources to be incorporated on the fly. While Condor is able to manage compute jobs remotely through Globus and other grid-computing protocols, dynamic deployment of Condor makes it possible to go one step further. Condor remotely installs and runs portions of itself. This process of Condor gliding in to inhabit computing resources on demand leverages the lowest common denominator of grid middleware systems, simple program execution, to bind together resources in a heterogeneous computing grid, with different management policies and different job execution methods, into a full-fledged Condor system.

The mobility of Condor services also benefits from the development of Condor-C, which provides a richer tool set for interlinking Condor-managed computers. Condor-C is a protocol that allows one Condor scheduler to delegate jobs to another Condor scheduler. The second scheduler could be at a remote site and/or an entry point into a restricted network. Delegating details of managing a job achieves greater flexibility with respect to network architecture, as well as fault tolerance and scalability. In the context of glide in deployments, the beach-head for each compute site is a dynamically deployed Condor scheduler which then serves as a target for Condor-C traffic.

In general, the mobility of the Condor scheduler and job execution agents, and the flexibility in how these are interconnected provide a uniform and feature-rich platform that can expand onto diverse resources and environments when the user requires it.

5.6 The Condor Job Router

The Condor Job Router is an add-on to the *condor_schedd* that transforms jobs from one type into another according to a configurable policy. This process of transforming the jobs is called *job routing*.

One example of how the Job Router can be used is for the task of sending excess jobs to one or more remote grid sites. The Job Router can transform the jobs such as vanilla universe jobs into grid universe jobs that use any of the grid types supported by Condor. The rate at which jobs are routed can be matched roughly to the rate at which the site is able to start running them. This makes it possible to balance a large work flow across multiple grid sites, a local Condor pool, and any flocked Condor pools, without having to guess in advance how quickly jobs will run and complete in each of the different sites.

Job Routing is most appropriate for high throughput work flows, where there are many more jobs than computers, and the goal is to keep as many of the computers busy as possible. Job Routing is less suitable when there are a small number of jobs, and the scheduler needs to choose the best place for each job, in order to finish them as quickly as possible. The Job Router does not know which site will run the jobs faster, but it can decide whether to send more jobs to a site, based on whether jobs already submitted to that site are sitting idle or not, as well as whether the site has experienced recent job failures.

5.6.1 Routing Mechanism

The *condor_job_router* daemon and configuration determine a policy for which jobs may be transformed and sent to grid sites. A job is transformed into a grid universe job by making a copy of the original job ClassAd, modifying some attributes of the job. The copy is called the routed copy, and it shows up in the job queue under a new job id.

Until the routed copy finishes or is removed, the original copy of the job passively mirrors the state of the routed job. During this time, the original job is not available for matchmaking, because it is tied to the routed copy. The original jobs also does not evaluate periodic expressions, such as `PeriodicHold`. Periodic expressions are evaluated for the routed copy. When the routed copy completes, the original job ClassAd is updated such that it reflects the final status of the job. If the routed copy is removed, the original job returns to the normal idle state, and is available for matchmaking or rerouting. If, instead, the original job is removed or goes on hold, the routed copy is removed.

The *condor_job_router* daemon utilizes a *routing table*, in which a ClassAd describes each site to where jobs may be sent. The routing table is given in the New ClassAd language, as currently used by Condor internally. A good place to learn about the syntax of New ClassAds is the Informal Language Description in the C++ ClassAds tutorial: http://www.cs.wisc.edu/condor/classad/c++tut.html.

Two essential differences distinguish the New ClassAd language from the current one. In the

New ClassAd language, each ClassAd is surrounded by square brackets. And, in the New ClassAd language, each assignment statement ends with a semicolon. When the New ClassAd is embedded in a Condor configuration file, it may appear all on a single line, but the readability is often improved by inserting line continuation characters after each assignment statement. This is done in the examples. Unfortunately, this makes the insertion of comments into the configuration file awkward, because of the interaction between comments and line continuation characters in configuration files. An alternative is to use C-style comments (`/* ...*/`). Another alternative is to read in the routing table entries from a separate file, rather than embedding them in the Condor configuration file.

5.6.2 Job Submission with Job Routing Capability

If Job Routing is set up, then the following items ought to be considered for jobs to have the necessary prerequisites to be considered for routing.

- Jobs appropriate for routing to the grid must not rely on access to a shared file system, or other services that are only available on the local pool. The job will use Condor's file transfer mechanism, rather than relying on a shared file system to access input files and write output files. In the submit description file, to enable file transfer, there will be a set of commands similar to

  ```
  should_transfer_files = YES
  when_to_transfer_output = ON_EXIT
  transfer_input_files = input1, input2
  transfer_output_files = output1, output2
  ```

  Vanilla universe jobs and most types of grid universe jobs differ in the set of files transferred back when the job completes. Vanilla universe jobs transfer back all files created or modified, while all grid universe jobs except for Condor-C only transfer back the **output** file, as well as those explicitly listed with **transfer_output_files**. Therefore, when routing jobs to grid universes other than Condor-C, it is important to explicitly specify all output files that must be transferred upon job completion.

  An additional difference between the vanilla universe jobs and **gt2** grid universe jobs is that **gt2** jobs do not return any information about the job's exit status. The exit status as reported in the job ClassAd and user log are always 0. Therefore, jobs that may be routed to a **gt2** grid site must not rely upon a non-zero job exit status.

- One configuration for routed jobs requires the jobs to identify themselves as candidates for Job Routing. This may be accomplished by inventing a ClassAd attribute that the configuration utilizes in setting the policy for job identification, and the job defines this attribute to identify itself. If the invented attribute is called `WantJobRouter`, then the job identifies itself as a job that may be routed by placing in the submit description file:

  ```
  +WantJobRouter = True
  ```

This implementation can be taken further, allowing the job to first be rejected within the local pool, before being a candidate for Job Routing:

```
+WantJobRouter = LastRejMatchTime =!= UNDEFINED
```

- As appropriate to the potential grid site, create a grid proxy, and specify it in the submit description file:

  ```
  x509userproxy = /tmp/x509up_u275
  ```

  This is not necessary if the *condor_job_router* daemon is configured to add a grid proxy on behalf of jobs.

Job submission does not change for jobs that may be routed.

```
$ condor_submit job1.sub
```

where `job1.sub` might contain:

```
universe = vanilla
executable = my_executable
output = job1.stdout
error = job1.stderr
log = job1.ulog
should_transfer_files = YES
when_to_transfer_output = ON_EXIT
+WantJobRouter = LastRejMatchTime =!= UNDEFINED
x509userproxy = /tmp/x509up_u275
queue
```

The status of the job may be observed as with any other Condor job, for example by looking in the job's log file. Before the job completes, *condor_q* shows the job's status. Should the job become routed, a second job will enter the job queue. This is the routed copy of the original job. The command *condor_router_q* shows a more specialized view of routed jobs, as this example shows:

```
$ condor_router_q -S
   JOBS ST Route      GridResource
     40  I Site1      site1.edu/jobmanager-condor
     10  I Site2      site2.edu/jobmanager-pbs
      2  R Site3      condor submit.site3.edu condor.site3.edu
```

condor_router_history summarizes the history of routed jobs, as this example shows:

```
$ condor_router_history
Routed job history from 2007-06-27 23:38 to 2007-06-28 23:38

Site            Hours    Jobs    Runs
                       Completed Aborted
-----------------------------------------------------
Site1              10       2     0
Site2               8       2     1
Site3              40       6     0
-----------------------------------------------------
TOTAL              58      10     1
```

5.6.3 An Example Configuration

The following sample configuration sets up potential job routing to three routes (grid sites). Definitions of the configuration variables specific to the Job Router are in section 3.3.21. One route is a Condor site accessed via the Globus gt2 protocol. A second route is a PBS site, also accessed via Globus gt2. The third site is a Condor site accessed by Condor-C. The *condor_job_router* daemon does not know which site will be best for a given job. The policy implemented in this sample configuration stops sending more jobs to a site, if ten jobs that have already been sent to that site are idle.

These configuration settings belong in the local configuration file of the machine where jobs are submitted. Check that the machine can successfully submit grid jobs before setting up and using the Job Router. Typically, the single required element that needs to be added for GSI authentication is an X.509 trusted certification authority directory, in a place recognized by Condor (for example, `/etc/grid-security/certificates`). The VDT (http://vdt.cs.wisc.edu) project provides a convenient way to set up and install a trusted CAs, if needed.

```
# These settings become the default settings for all routes
JOB_ROUTER_DEFAULTS = \
  [ \
    requirements=target.WantJobRouter is True; \
    MaxIdleJobs = 10; \
    MaxJobs = 200; \
\
    /* now modify routed job attributes */ \
    /* remove routed job if it goes on hold or stays idle for over 6 hours */ \
    set_PeriodicRemove = JobStatus == 5 || \
                         (JobStatus == 1 && (CurrentTime - QDate) > 3600*6); \
    delete_WantJobRouter = true; \
    set_requirements = true; \
  ]

# This could be made an attribute of the job, rather than being hard-coded
ROUTED_JOB_MAX_TIME = 1440

# Now we define each of the routes to send jobs on
```

```
JOB_ROUTER_ENTRIES = \
   [ GridResource = "gt2 site1.edu/jobmanager-condor"; \
     name = "Site 1"; \
   ] \
   [ GridResource = "gt2 site2.edu/jobmanager-pbs"; \
     name = "Site 2"; \
     set_GlobusRSL = "(maxwalltime=$(ROUTED_JOB_MAX_TIME))(jobType=single)"; \
   ] \
   [ GridResource = "condor submit.site3.edu condor.site3.edu"; \
     name = "Site 3"; \
     set_remote_jobuniverse = 5; \
   ]


# Reminder: you must restart Condor for changes to DAEMON_LIST to take effect.
DAEMON_LIST = $(DAEMON_LIST) JOB_ROUTER

# For testing, set this to a small value to speed things up.
# Once you are running at large scale, set it to a higher value
# to prevent the JobRouter from using too much cpu.
JOB_ROUTER_POLLING_PERIOD = 10

#It is good to save lots of schedd queue history
#for use with the router_history command.
MAX_HISTORY_ROTATIONS = 20
```

5.6.4 Routing Table Entry ClassAd Attributes

The conversion of a job to a routed copy may require the job ClassAd to be modified. The Routing Table specifies attributes of the different possible routes and it may specify specific modifications that should be made to the job when it is sent along a specific route.

The following attributes and instructions for modifying job attributes may appear in a Routing Table entry.

GridResource Specifies the value for the `GridResource` attribute that will be inserted into the routed copy of the job's ClassAd.

Name An optional identifier that will be used in log messages concerning this route. If no name is specified, the default used will be the value of `GridResource`. The *condor_job_router* distinguishes routes and advertises statistics based on this attribute's value.

Requirements A `Requirements` expression in New ClassAd syntax that identifies jobs that may be matched to the route. Note that, as with all settings, requirements specified in the configuration variable `JOB_ROUTER_ENTRIES` override the setting of `JOB_ROUTER_DEFAULTS`. To specify global requirements that are not overridden by `JOB_ROUTER_ENTRIES`, use `JOB_ROUTER_SOURCE_JOB_CONSTRAINT`.

MaxJobs An integer maximum number of jobs permitted on the route at one time. The default is 100.

MaxIdleJobs An integer maximum number of routed jobs in the idle state. At or above this value, no more jobs will be sent to this site. This is intended to prevent too many jobs from being sent to sites which are too busy to run them. If the value set for this attribute is too small, the rate of job submission to the site will slow, because the *condor_job_router* daemon will submit jobs up to this limit, wait to see some of the jobs enter the running state, and then submit more. The disadvantage of setting this attribute's value too high is that a lot of jobs may be sent to a site, only to site idle for hours or days. The default value is 50.

FailureRateThreshold A maximum tolerated rate of job failures. Failure is determined by the expression sets for the attribute `JobFailureTest` expression. The default threshold is 0.03 jobs/second. If the threshold is exceeded, submission of new jobs is throttled until jobs begin succeeding, such that the failure rate is less than the threshold. This attribute implements *black hole throttling*, such that a site at which jobs are sent only to fail (a black hole) receives fewer jobs.

JobFailureTest An expression in New ClassAds syntax evaluated for each job that finishes, to determine whether it was a failure. The default value if no expression is defined assumes all jobs are successful. Routed jobs that are removed are considered to be failures. An example expression to treat all jobs running for less than 30 minutes as failures is `target.RemoteWallClockTime < 1800`. A more flexible expression might reference a property or expression of the job that specifies a failure condition specific to the type of job.

TargetUniverse An integer value specifying the desired universe for the routed copy of the job. The default value is 9, which is the **grid** universe.

UseSharedX509UserProxy A boolean expression in New ClassAds syntax, that when `True` causes the value of `SharedX509UserProxy` to be the X.509 user proxy for the routed job. Note that if the *condor_job_router* daemon is running as root, the copy of this file that is given to the job will have its ownership set to that of the user running the job. This requires the trust of the user. It is therefore recommended to avoid this mechanism when possible. Instead, require users to submit jobs with `X509UserProxy` set in the submit description file. If this feature is needed, use the boolean expression to only allow specific values of `target.Owner` to use this shared proxy file. The shared proxy file should be owned by the `condor` user. Currently, to use a shared proxy, the job must also turn on sandboxing by having the attribute `JobShouldBeSandboxed`.

SharedX509UserProxy A string representing file containing the X.509 user proxy for the routed job.

JobShouldBeSandboxed A boolean expression in New ClassAd syntax, that when `True` causes the created copy of the job to be sandboxed. A copy of the input files will be placed in the *condor_schedd* daemon's spool area for the target job, and when the job runs, the output will be staged back into the spool area. Once all of the output has been successfully staged back, it will be copied again, this time from the spool area of the sandboxed job back to the original job's output locations. By default, sandboxing is turned off. Only to turn it on if using a shared X.509 user proxy or if direct staging of remote output files back to the final output locations is not desired.

OverrideRoutingEntry A boolean value that when `True`, indicates that this entry in the routing table replaces any previous entry in the table with the same name. When `False`, it indicates that if there is a previous entry by the same name, the previous entry should be retained and this entry should be ignored. The default value is `True`.

Set_<ATTR> Sets the value of `<ATTR>` in the routed job ClassAd to the specified value. An example of an attribute that might be set is `PeriodicRemove`. For example, if the routed job goes on hold or stays idle for too long, remove it and return the original copy of the job to a normal state.

Eval_Set_<ATTR> Defines an expression written in New ClassAd syntax. The expression is evaluated, and the resulting value sets the value of the routed copy's job ClassAd attribute `<ATTR>`. An expected usage is where a New ClassAd expression evaluation is required. Recall that the *condor_job_router* daemon evaluates using New ClassAd syntax, but the routed copy is represented by the current ClassAd language.

Copy_<ATTR> Defined with the name of a routed copy ClassAd attribute. Copies the value of `<ATTR>` from the original job ClassAd into the specified attribute named of the routed copy. Useful to save the value of an expression, before replacing it with something else that references the original expression.

Delete_<ATTR> Deletes `<ATTR>` from the routed copy ClassAd. A value assigned to this attribute in the routing table entry is ignored.

5.6.5 Example: constructing the routing table from ReSS

The Open Science Grid has a service called ReSS (Resource Selection Service). It presents grid sites as ClassAds in a Condor collector. This example builds a routing table from the site ClassAds in the ReSS collector.

Using `JOB_ROUTER_ENTRIES_CMD` , we tell the *condor_job_router* daemon to call a simple script which queries the collector and outputs a routing table. The script, called `osg_ress_routing_table.sh`, is just this:

```
#!/bin/sh

# you _MUST_ change this:
export condor_status=/path/to/condor_status
# if no command line arguments specify -pool, use this:
export _CONDOR_COLLECTOR_HOST=osg-ress-1.fnal.gov

$condor_status -format '[ ' BeginAd \
               -format 'GridResource = "gt2 %s"; ' GlueCEInfoContactString \
      -format ']\n' EndAd "$@" | uniq
```

Save this script to a file and make sure the permissions on the file mark it as executable. Test this script by calling it by hand before trying to use it with the *condor_job_router* daemon. You may supply additional arguments such as **-constraint** to limit the sites which are returned.

Once you are satisfied that the routing table constructed by the script is what you want, configure the *condor_job_router* daemon to use it:

```
# command to build the routing table
JOB_ROUTER_ENTRIES_CMD = /path/to/osg_ress_routing_table.sh <extra arguments>

# how often to rebuild the routing table:
JOB_ROUTER_ENTRIES_REFRESH = 3600
```

Using the example configuration, use the above settings to replace `JOB_ROUTER_ENTRIES` . Or, leave `JOB_ROUTER_ENTRIES` there and have a routing table containing entries from both sources. When you restart or reconfigure the *condor_job_router* daemon, you should see messages in the Job Router's log indicating that it is adding more routes to the table.

CHAPTER

SIX

Platform-Specific Information

The Condor Team strives to make Condor work the same way across all supported platforms. However, because Condor is a very low-level system which interacts closely with the internals of the operating systems on which it runs, this goal is not always possible to achieve. The following sections provide detailed information about using Condor on different computing platforms and operating systems.

6.1 Linux

This section provides information specific to the Linux port of Condor. Linux is a difficult platform to support. It changes very frequently, and Condor has some extremely system-dependent code (for example, the checkpointing library).

Condor is sensitive to changes in the following elements of the system:

- The kernel version
- The version of the GNU C library (glibc)
- the version of GNU C Compiler (GCC) used to build and link Condor jobs (this only matters for Condor's Standard universe which provides checkpointing and remote system calls)

The Condor Team tries to provide support for various releases of the distribution of Linux. Red Hat is probably the most popular Linux distribution, and it provides a common set of versions for the above system components at which Condor can aim support. Condor will often work with Linux distributions other than Red Hat (for example, Debian or SuSE) that have the same versions of the

above components. However, we do not usually test Condor on other Linux distributions and we do not provide any guarantees about this.

New releases of Red Hat usually change the versions of some or all of the above system-level components. A version of Condor that works with one release of Red Hat might not work with newer releases. The following sections describe the details of Condor's support for the currently available versions of Red Hat Linux on x86 architecture machines.

6.1.1 Linux Kernel-specific Information

Distributions that rely on the Linux 2.4.x and all Linux 2.6.x kernels through version 2.6.10 do not modify the `atime` of the input device file. This leads to difficulty when Condor is run using one of these kernels. The problem manifests itself in that Condor cannot properly detect keyboard or mouse activity. Therefore, using the activity in policy setting cannot signal that Condor should stop running a job on a machine.

Condor version 6.6.8 implements a workaround for PS/2 devices. A better fix is the Linux 2.6.10 kernel patch linked to from the directions posted at http://www.cs.wisc.edu/condor/kernel.patch.html. This patch works better for PS/2 devices, and may also work for USB devices. A future version of Condor will implement better recognition of USB devices, such that the kernel patch will also definitively work for USB devices.

Condor additionally has problems running on some older Xen kernels, which interact badly with assumptions made by the *condor_procd* daemon. See the FAQ entry in section 7.7 for details.

6.1.2 Address Space Randomization

Modern versions of Red Hat and Fedora do address space randomization, which randomizes the memory layout of a process to reduce the possibility of security exploits. This makes it impossible for standard universe jobs to resume execution using a checkpoint. When starting or resuming a standard universe job, Condor disables the randomization.

To run a binary compiled with *condor_compile* in standalone mode, either initially or in resumption mode, manually disable the address space randomization by modifying the command line. For a 32-bit architecture, assuming a Condor-linked binary called *myapp*, invoke the standalone executable with:

```
setarch i386 -L -R ./myapp
```

For a 64-bit architecture, the resumption command will be:

```
setarch x86_64 -L -R ./myapp
```

Some applications will also need the **-B** option.

The command to resume execution using the checkpoint must also disable address space randomization, as the 32-bit architecture example:

```
setarch i386 -L -R myapp -_condor_restart myapp.ckpt
```

6.2 Microsoft Windows

Windows is a strategic platform for Condor, and therefore we have been working toward a complete port to Windows. Our goal is to make Condor every bit as capable on Windows as it is on Unix – or even more capable.

Porting Condor from Unix to Windows is a formidable task, because many components of Condor must interact closely with the underlying operating system. Provided is a clipped version of Condor for Windows. A clipped version is one in which there is no checkpointing and there are no remote system calls.

This section contains additional information specific to running Condor on Windows. In order to effectively use Condor, first read the overview chapter (section 1.1) and the user's manual (section 2.1). If administrating or customizing the policy and set up of Condor, also read the administrator's manual chapter (section 3.1). After reading these chapters, review the information in this chapter for important information and differences when using and administrating Condor on Windows. For information on installing Condor for Windows, see section 3.2.5.

6.2.1 Limitations under Windows

In general, this release for Windows works the same as the release of Condor for Unix. However, the following items are not supported in this version:

- The standard job universe is not present. This means transparent process checkpoint/migration and remote system calls are not supported.
- For **grid** universe jobs, the only supported grid type is **condor**.
- Accessing files via a network share that requires a Kerberos ticket (such as AFS) is not yet supported.

6.2.2 Supported Features under Windows

Except for those items listed above, most everything works the same way in Condor as it does in the Unix release. This release is based on the Condor Version 7.6.0 source tree, and thus the feature set is the same as Condor Version 7.6.0 for Unix. For instance, all of the following work in Condor:

- The ability to submit, run, and manage queues of jobs running on a cluster of Windows machines.
- All tools such as *condor_q*, *condor_status*, *condor_userprio*, are included. Only *condor_compile* is *not* included.
- The ability to customize job policy using ClassAds. The machine ClassAds contain all the information included in the Unix version, including current load average, RAM and virtual memory sizes, integer and floating-point performance, keyboard/mouse idle time, etc. Likewise, job ClassAds contain a full complement of information, including system dependent entries such as dynamic updates of the job's image size and CPU usage.
- Everything necessary to run a Condor central manager on Windows.
- Security mechanisms.
- Support for SMP machines.
- Condor for Windows can run jobs at a lower operating system priority level. Jobs can be suspended, soft-killed by using a WM_CLOSE message, or hard-killed automatically based upon policy expressions. For example, Condor can automatically suspend a job whenever keyboard/mouse or non-Condor created CPU activity is detected, and continue the job after the machine has been idle for a specified amount of time.
- Condor correctly manages jobs which create multiple processes. For instance, if a Condor job spawns multiple processes and Condor needs to kill the job, all processes created by the job will be terminated.
- In addition to interactive tools, users and administrators can receive information from Condor by e-mail (standard SMTP) and/or by log files.
- Condor includes a friendly GUI installation and set up program, which can perform a full install or deinstall of Condor. Information specified by the user in the set up program is stored in the system registry. The set up program can update a current installation with a new release using a minimal amount of effort.
- Condor can give a job access to the running user's Registry hive.

6.2.3 Secure Password Storage

In order for Condor to operate properly, it must at times be able to act on behalf of users who submit jobs. This is required on submit machines, so that Condor can access a job's input files, create and access the job's output files, and write to the job's log file from within the appropriate security context. On Unix systems, arbitrarily changing what user Condor performs its actions as is easily done when Condor is started with root privileges. On Windows, however, performing an action as a particular user or on behalf of a particular user requires knowledge of that user's password, even when running at the maximum privilege level. Condor provides secure password storage through the use of the *condor_store_cred* tool. Passwords managed by Condor are encrypted and stored in

a secure location within the Windows registry. When Condor needs to perform an action as or on behalf of a particular user, it uses the securely stored password to do so. This implies that a password is stored for every user that will submit jobs from the Windows submit machine.

A further feature permits Condor to execute the job itself under the security context of its submitting user, specifying the **run_as_owner** command in the job's submit description file. With this feature, it is necessary to configure and run a centralized *condor_credd* daemon to manage the secure password storage. This makes each user's password available, via an encrypted connection to the *condor_credd*, to any execute machine that may need it.

By default, the secure password store for a submit machine when no *condor_credd* is running is managed by the *condor_schedd*. This approach works in environments where the user's password is only needed on the submit machine.

6.2.4 Executing Jobs as the Submitting User

By default, Condor executes jobs on Windows using dedicated run accounts that have minimal access rights and privileges, and which are recreated for each new job. As an alternative, Condor can be configured to allow users to run jobs using their Windows login accounts. This may be useful if jobs need access to files on a network share, or to other resources that are not available to the low-privilege run account.

This feature requires use of a *condor_credd* daemon for secure password storage and retrieval. With the *condor_credd* daemon running, the user's password must be stored, using the *condor_store_cred* tool. Then, a user that wants a job to run using their own account places into the job's submit description file

```
run_as_owner = True
```

6.2.5 The condor_credd Daemon

The *condor_credd* daemon manages secure password storage. A single running instance of the *condor_credd* within a Condor pool is necessary in order to provide the feature described in section 6.2.4, where a job runs as the submitting user, instead of as a temporary user that has strictly limited access capabilities.

It is first necessary to select the single machine on which to run the *condor_credd*. Often, the machine acting as the pool's central manager is a good choice. An important restriction, however, is that the *condor_credd* host must be a machine running Windows.

All configuration settings necessary to enable the *condor_credd* are contained in the example file `etc\condor_config.local.credd` from the Condor distribution. Copy these settings into a local configuration file for the machine that will run the *condor_credd*. Run `condor_restart` for these new settings to take effect, then verify (via Task Manager) that a *condor_credd* process is running.

A second set of configuration variables specify security for the communication among Condor daemons. These variables must be set for all machines in the pool. The following example settings are in the comments contained in the `etc\condor_config.local.credd` example file. These sample settings rely on the `PASSWORD` method for authentication among daemons, including communication with the *condor_credd* daemon. The `LOCAL_CREDD` variable must be customized to point to the machine hosting the *condor_credd* and the `ALLOW_CONFIG` variable will be customized, if needed, to refer to an administrative account that exists on all Condor nodes.

```
CREDD_HOST = credd.cs.wisc.edu
CREDD_CACHE_LOCALLY = True

STARTER_ALLOW_RUNAS_OWNER = True

ALLOW_CONFIG = Administrator@*
SEC_CLIENT_AUTHENTICATION_METHODS = NTSSPI, PASSWORD
SEC_CONFIG_NEGOTIATION = REQUIRED
SEC_CONFIG_AUTHENTICATION = REQUIRED
SEC_CONFIG_ENCRYPTION = REQUIRED
SEC_CONFIG_INTEGRITY = REQUIRED
```

In order for `PASSWORD` authenticated communication to work, a *pool password* must be chosen and distributed. The chosen pool password must be stored identically for each machine. The pool password first should be stored on the *condor_credd* host, then on the other machines in the pool.

To store the pool password on a Windows machine, run

```
condor_store_cred add -c
```

when logged in with the administrative account on that machine, and enter the password when prompted. If the administrative account is shared across all machines, that is if it is a domain account or has the same password on all machines, logging in separately to each machine in the pool can be avoided. Instead, the pool password can be securely pushed out for each Windows machine using a command of the form

```
condor_store_cred add -c -n exec01.cs.wisc.edu
```

Once the pool password is distributed, but before submitting jobs, all machines must reevaluate their configuration, so execute

```
condor_reconfig -all
```

from the central manager. This will cause each execute machine to test its ability to authenticate with the *condor_credd*. To see whether this test worked for each machine in the pool, run the command

```
condor_status -f "%s\t" Name -f "%s\n" ifThenElse(isUndefined(LocalCredd),\"UNDEF\",LocalCredd)
```

Any rows in the output with the UNDEF string indicate machines where secure communication is not working properly. Verify that the pool password is stored correctly on these machines.

6.2.6 Executing Jobs with the User's Profile Loaded

Condor can be configured when using dedicated run accounts, to load the account's profile. A user's profile includes a set of personal directories and a registry hive loaded under HKEY_CURRENT_USER.

This may be useful if the job requires direct access to the user's registry entries. It also may be useful when the job requires an application, and the application requires registry access. This feature is always enabled on the *condor_startd*, but it is limited to the dedicated run account. For security reasons, the profiles are removed after the job has completed and exited. This ensures that malicious jobs cannot discover what any previous job has done, nor sabotage the registry for future jobs. It also ensures the next job has a fresh registry hive.

A user that then wants a job to run with a profile uses the **load_profile** command in the job's submit description file:

```
load_profile = True
```

This feature is currently not compatible with **run_as_owner**, and will be ignored if both are specified.

6.2.7 Using Windows Scripts as Job Executables

Condor has added support for scripting jobs on Windows. Previously, Condor jobs on Windows were limited to executables or batch files. With this new support, Condor determines how to interpret the script using the file name's extension. Without a file name extension, the file will be treated as it has been in the past: as a Windows executable.

This feature may not require any modifications to Condor's configuration. An example that does not require administrative intervention are Perl scripts using *ActivePerl*.

Windows Scripting Host scripts do require configuration to work correctly. The configuration variables set values to be used in registry look up, which results in a command that invokes the correct interpreter, with the correct command line arguments for the specific scripting language. In Microsoft nomenclature, *verbs* are actions that can be taken upon a given a file. The familiar examples of **Open**, **Print**, and **Edit**, can be found on the context menu when a user right clicks on a file. The command lines to be used for each of these verbs are stored in the registry under the HKEY_CLASSES_ROOT hive. In general, a registry look up uses the form:

```
HKEY_CLASSES_ROOT\<FileType>\Shell\<OpenVerb>\Command
```

Within this specification, <FileType> is the name of a file type (and therefore a scripting language), and is obtained from the file name extension. <OpenVerb> identifies the verb, and is obtained from the Condor configuration.

The Condor configuration sets the selection of a verb, to aid in the registry look up. The file name extension sets the name of the Condor configuration variable. This variable name is of the form:

```
OPEN_VERB_FOR_<EXT>_FILES
```

<EXT> represents the file name extension. The following configuration example uses the Open2 verb for a *Windows Scripting Host* registry look up for several scripting languages:

```
OPEN_VERB_FOR_JS_FILES  = Open2
OPEN_VERB_FOR_VBS_FILES = Open2
OPEN_VERB_FOR_VBE_FILES = Open2
OPEN_VERB_FOR_JSE_FILES = Open2
OPEN_VERB_FOR_WSF_FILES = Open2
OPEN_VERB_FOR_WSH_FILES = Open2
```

In this example, Condor specifies the Open2 verb, instead of the default Open verb, for a script with the file name extension of wsh. The *Windows Scripting Host*'s Open2 verb allows standard input, standard output, and standard error to be redirected as needed for Condor jobs.

A common difficulty is encountered when a script interpreter requires access to the user's registry. Note that the user's registry is different than the root registry. If not given access to the user's registry, some scripts, such as *Windows Scripting Host* scripts, will fail. The failure error message appears as:

```
CScript Error: Loading your settings failed. (Access is denied.)
```

The fix for this error is to give explicit access to the submitting user's registry hive. This can be accomplished with the addition of the **load_profile** command in the job's submit description file:

```
load_profile = True
```

With this command, there should be no registry access errors. This command should also work for other interpreters. Note that not all interpreters will require access. For example, *ActivePerl* does not by default require access to the user's registry hive.

6.2.8 Details on how Condor for Windows starts/stops a job

This section provides some details on how Condor starts and stops jobs. This discussion is geared for the Condor administrator or advanced user who is already familiar with the material in the Ad-

ministrator's Manual and wishes to know detailed information on what Condor does when starting and stopping jobs.

When Condor is about to start a job, the *condor_startd* on the execute machine spawns a *condor_starter* process. The *condor_starter* then creates:

1. a run account on the machine with a login name of "condor-reuse-slotX", where X is the slot number of the *condor_starter*. This account is added to group Users. This step is skipped if the job is to be run using the submitting user's account (see section 6.2.4).
2. a new temporary working directory for the job on the execute machine. This directory is named "dir_XXX", where XXX is the process ID of the *condor_starter*. The directory is created in the `$(EXECUTE)` directory as specified in Condor's configuration file. Condor then grants write permission to this directory for the user account newly created for the job.
3. a new, non-visible Window Station and Desktop for the job. Permissions are set so that only the account that will run the job has access rights to this Desktop. Any windows created by this job are not seen by anyone; the job is run in the background. (Note: Setting `USE_VISIBLE_DESKTOP` to True will allow the job to access the default desktop instead of a newly created one.)

Next, the *condor_starter* (called the starter) contacts the *condor_shadow* (called the shadow) process, which is running on the submitting machine, and pulls over the job's executable and input files. These files are placed into the temporary working directory for the job. After all files have been received, the starter spawns the user's executable. Its current working directory set to the temporary working directory (that is, `$(EXECUTE)`/dir_XXX, where XXX is the process id of the *condor_starter* daemon).

While the job is running, the starter closely monitors the CPU usage and image size of all processes started by the job. Every 20 minutes the starter sends this information, along with the total size of all files contained in the job's temporary working directory, to the shadow. The shadow then inserts this information into the job's ClassAd so that policy and scheduling expressions can make use of this dynamic information.

If the job exits of its own accord (that is, the job completes), the starter first terminates any processes started by the job which could still be around if the job did not clean up after itself. The starter examines the job's temporary working directory for any files which have been created or modified and sends these files back to the shadow running on the submit machine. The shadow places these files into the **initialdir** specified in the submit description file; if no **initialdir** was specified, the files go into the directory where the user invoked *condor_submit*. Once all the output files are safely transferred back, the job is removed from the queue. If, however, the *condor_startd* forcibly kills the job before all output files could be transferred, the job is not removed from the queue but instead switches back to the Idle state.

If the *condor_startd* decides to vacate a job prematurely, the starter sends a WM_CLOSE message to the job. If the job spawned multiple child processes, the WM_CLOSE message is only sent to the parent process (that is, the one started by the starter). The WM_CLOSE message is the preferred way to terminate a process on Windows, since this method allows the job to cleanup and

free any resources it may have allocated. When the job exits, the starter cleans up any processes left behind. At this point, if **transfer_files** is set to *ONEXIT* (the default) in the job's submit description file, the job switches from states, from Running to Idle, and no files are transferred back. If **transfer_files** is set to *ALWAYS*, then any files in the job's temporary working directory which were changed or modified are first sent back to the submitting machine. But this time, the shadow places these so-called intermediate files into a subdirectory created in the `$(SPOOL)` directory on the submitting machine (`$(SPOOL)` is specified in Condor's configuration file). The job is then switched back to the Idle state until Condor finds a different machine on which to run. When the job is started again, Condor places into the job's temporary working directory the executable and input files as before, *plus* any files stored in the submit machine's `$(SPOOL)` directory for that job.

NOTE: A Windows console process can intercept a WM_CLOSE message via the Win32 SetConsoleCtrlHandler() function if it needs to do special cleanup work at vacate time; a WM_CLOSE message generates a CTRL_CLOSE_EVENT. See SetConsoleCtrlHandler() in the Win32 documentation for more info.

NOTE: The default handler in Windows for a WM_CLOSE message is for the process to exit. Of course, the job could be coded to ignore it and not exit, but eventually the *condor_startd* will become impatient and hard-kill the job (if that is the policy desired by the administrator).

Finally, after the job has left and any files transferred back, the starter deletes the temporary working directory, the temporary account (if one was created), the WindowStation, and the Desktop before exiting. If the starter should terminate abnormally, the *condor_startd* attempts the clean up. If for some reason the *condor_startd* should disappear as well (that is, if the entire machine was power-cycled hard), the *condor_startd* will clean up when Condor is restarted.

6.2.9 Security Considerations in Condor for Windows

On the execute machine (by default), the user job is run using the access token of an account dynamically created by Condor which has bare-bones access rights and privileges. For instance, if your machines are configured so that only Administrators have write access to `C:\WINNT`, then certainly no Condor job run on that machine would be able to write anything there. The only files the job should be able to access on the execute machine are files accessible by the Users and Everyone groups, and files in the job's temporary working directory. Of course, if the job is configured to run using the account of the submitting user (as described in section 6.2.4), it will be able to do anything that the user is able to do on the execute machine it runs on.

On the submit machine, Condor impersonates the submitting user, therefore the File Transfer mechanism has the same access rights as the submitting user. For example, say only Administrators can write to `C:\WINNT` on the submit machine, and a user gives the following to *condor_submit* :

```
executable = mytrojan.exe
initialdir = c:\winnt
output = explorer.exe
queue
```

Unless that user is in group Administrators, Condor will not permit `explorer.exe` to be overwritten.

If for some reason the submitting user's account disappears between the time *condor_submit* was run and when the job runs, Condor is not able to check and see if the now-defunct submitting user has read/write access to a given file. In this case, Condor will ensure that group "Everyone" has read or write access to any file the job subsequently tries to read or write. This is in consideration for some network setups, where the user account only exists for as long as the user is logged in.

Condor also provides protection to the job queue. It would be bad if the integrity of the job queue is compromised, because a malicious user could remove other user's jobs or even change what executable a user's job will run. To guard against this, in Condor's default configuration all connections to the *condor_schedd* (the process which manages the job queue on a given machine) are authenticated using Windows' eSSPI security layer. The user is then authenticated using the same challenge-response protocol that Windows uses to authenticate users to Windows file servers. Once authenticated, the only users allowed to edit job entry in the queue are:

1. the user who originally submitted that job (i.e. Condor allows users to remove or edit their own jobs)
2. users listed in the `condor_config` file parameter `QUEUE_SUPER_USERS`. In the default configuration, only the "SYSTEM" (LocalSystem) account is listed here.

WARNING: Do not remove "SYSTEM" from `QUEUE_SUPER_USERS`, or Condor itself will not be able to access the job queue when needed. If the LocalSystem account on your machine is compromised, you have all sorts of problems!

To protect the actual job queue files themselves, the Condor installation program will automatically set permissions on the entire Condor release directory so that only Administrators have write access.

Finally, Condor has all the IP/Host-based security mechanisms present in the full-blown version of Condor. See section 3.6.9 starting on page 342 for complete information on how to allow/deny access to Condor based upon machine host name or IP address.

6.2.10 Network files and Condor

Condor can work well with a network file server. The recommended approach to having jobs access files on network shares is to configure jobs to run using the security context of the submitting user (see section 6.2.4). If this is done, the job will be able to access resources on the network in the same way as the user can when logged in interactively.

In some environments, running jobs as their submitting users is not a feasible option. This section outlines some possible alternatives. The heart of the difficulty in this case is that on the execute machine, Condor creates a temporary user that will run the job. The file server has never heard of this user before.

Choose one of these methods to make it work:

- METHOD A: access the file server as a different user via a net use command with a login and password
- METHOD B: access the file server as guest
- METHOD C: access the file server with a "NULL" descriptor
- METHOD D: create and have Condor use a special account
- METHOD E: use the contrib module from the folks at Bristol University

All of these methods have advantages and disadvantages.

Here are the methods in more detail:

METHOD A - access the file server as a different user via a net use command with a login and password

Example: you want to copy a file off of a server before running it....

```
@echo off
net use \\myserver\someshare MYPASSWORD /USER:MYLOGIN
copy \\myserver\someshare\my-program.exe
my-program.exe
```

The idea here is to simply authenticate to the file server with a different login than the temporary Condor login. This is easy with the "net use" command as shown above. Of course, the obvious disadvantage is this user's password is stored and transferred as clear text.

METHOD B - access the file server as guest

Example: you want to copy a file off of a server before running it as GUEST

```
@echo off
net use \\myserver\someshare
copy \\myserver\someshare\my-program.exe
my-program.exe
```

In this example, you'd contact the server MYSERVER as the Condor temporary user. However, if you have the GUEST account enabled on MYSERVER, you will be authenticated to the server as user "GUEST". If your file permissions (ACLs) are setup so that either user GUEST (or group EVERYONE) has access the share "someshare" and the directories/files that live there, you can use this method. The downside of this method is you need to enable the GUEST account on your file server. WARNING: This should be done *with extreme caution* and only if your file server is well protected behind a firewall that blocks SMB traffic.

METHOD C - access the file server with a "NULL" descriptor

One more option is to use NULL Security Descriptors. In this way, you can specify which shares are accessible by NULL Descriptor by adding them to your registry. You can then use the batch file wrapper like:

```
net use z: \\myserver\someshare /USER:""
z:\my-program.exe
```

so long as 'someshare' is in the list of allowed NULL session shares. To edit this list, run regedit.exe and navigate to the key:

```
HKEY_LOCAL_MACHINE\
   SYSTEM\
     CurrentControlSet\
       Services\
         LanmanServer\
           Parameters\
             NullSessionShares
```

and edit it. unfortunately it is a binary value, so you'll then need to type in the hex ASCII codes to spell out your share. each share is separated by a null (0x00) and the last in the list is terminated with two nulls.

although a little more difficult to set up, this method of sharing is a relatively safe way to have one quasi-public share without opening the whole guest account. you can control specifically which shares can be accessed or not via the registry value mentioned above.

METHOD D - create and have Condor use a special account

Create a permanent account (called condor-guest in this description) under which Condor will run jobs. On all Windows machines, and on the file server, create the condor-guest account.

On the network file server, give the condor-guest user permissions to access files needed to run Condor jobs.

Securely store the password of the condor-guest user in the Windows registry using *condor_store_cred* on all Windows machines.

Tell Condor to use the condor-guest user as the owner of jobs, when required. Details for this are in section 3.6.13.

METHOD E - access with the contrib module from Bristol

Another option: some hardcore Condor users at Bristol University developed their own module for starting jobs under Condor NT to access file servers. It involves storing submitting user's passwords on a centralized server. Below I have included the README from this contrib module, which

will soon appear on our website within a week or two. If you want it before that, let me know, and I could e-mail it to you.

Here is the README from the Bristol Condor contrib module:

```
README
Compilation Instructions
Build the projects in the following order

CondorCredSvc
CondorAuthSvc
Crun
Carun
AfsEncrypt
RegisterService
DeleteService
Only the first 3 need to be built in order. This just makes sure that the
RPC stubs are correctly rebuilt if required. The last 2 are only helper
applications to install/remove the services. All projects are Visual Studio
6 projects. The nmakefiles have been exported for each. Only the project
for Carun should need to be modified to change the location of the AFS
libraries if needed.

Details
CondorCredSvc
CondorCredSvc is a simple RPC service that serves the domain account
credentials. It reads the account name and password from the registry of
the machine it's running on. At the moment these details are stored in
clear text under the key

HKEY_LOCAL_MACHINE\Software\Condor\CredService

The account name and password are held in REG_SZ values "Account" and
"Password" respectively. In addition there is an optional REG_SZ value
"Port" which holds the clear text port number (e.g. "1234"). If this value
is not present the service defaults to using port 3654.

At the moment there is no attempt to encrypt the username/password when it
is sent over the wire - but this should be reasonably straightforward to
change. This service can sit on any machine so keeping the registry entries
secure ought to be fine. Certainly the ACL on the key could be set to only
allow administrators and SYSTEM access.

CondorAuthSvc and Crun
These two programs do the hard work of getting the job authenticated and
running in the right place. CondorAuthSvc actually handles the process
```

creation while Crun deals with getting the winstation/desktop/working directory and grabbing the console output from the job so that Condor's output handling mechanisms still work as advertised. Probably the easiest way to see how the two interact is to run through the job creation process:

The first thing to realize is that condor itself only runs Crun.exe. Crun treats its command line parameters as the program to really run. e.g. "Crun \\mymachine\myshare\myjob.exe" actually causes \\mymachine\myshare\myjob.exe to be executed in the context of the domain account served by CondorCredSvc. This is how it works:

When Crun starts up it gets its window station and desktop - these are the ones created by condor. It also gets its current directory - again already created by condor. It then makes sure that SYSTEM has permission to modify the DACL on the window station, desktop and directory. Next it creates a shared memory section and copies its environment variable block into it. Then, so that it can get hold of STDOUT and STDERR from the job it makes two named pipes on the machine it's running on and attaches a thread to each which just prints out anything that comes in on the pipe to the appropriate stream. These pipes currently have a NULL DACL, but only one instance of each is allowed so there shouldn't be any issues involving malicious people putting garbage into them. The shared memory section and both named pipes are tagged with the ID of Crun's process in case we're on a multi-processor machine that might be running more than one job. Crun then makes an RPC call to CondorAuthSvc to actually start the job, passing the names of the window station, desktop, executable to run, current directory, pipes and shared memory section (it only attempts to call CondorAuthSvc on the same machine as it is running on). If the jobs starts successfully it gets the process ID back from the RPC call and then just waits for the new process to finish before closing the pipes and exiting. Technically, it does this by synchronizing on a handle to the process and waiting for it to exit. CondorAuthSvc sets the ACL on the process to allow EVERYONE to synchronize on it.

[Technical note: Crun adds "C:\WINNT\SYSTEM32\CMD.EXE /C" to the start of the command line. This is because the process is created with the network context of the caller i.e. LOCALSYSTEM. Pre-pending cmd.exe gets round any unexpected "Access Denied" errors.]

If Crun gets a WM_CLOSE (CTRL_CLOSE_EVENT) while the job is running it attempts to stop the job, again with an RPC call to CondorAuthSvc passing the job's process ID.

CondorAuthSvc runs as a service under the LOCALSYSTEM account and does the work of starting the job. By default it listens on port 3655, but this can be changed by setting the optional REG_SZ value "Port" under the registry key

HKEY_LOCAL_MACHINE\Software\Condor\AuthService

(Crun also checks this registry key when attempting to contact CondorAuthSvc.) When it gets the RPC to start a job CondorAuthSvc first connects to the pipes for STDOUT and STDERR to prevent anyone else sending data to them. It also opens the shared memory section with the environment stored by Crun. It then makes an RPC call to CondorCredSvc (to get the name and password of the domain account) which is most likely running on another system. The location information is stored in the registry under the key

HKEY_LOCAL_MACHINE\Software\Condor\CredService

The name of the machine running CondorCredSvc must be held in the REG_SZ value "Host". This should be the fully qualified domain name of the machine. You can also specify the optional "Port" REG_SZ value in case you are running CondorCredSvc on a different port.

Once the domain account credentials have been received the account is logged on through a call to LogonUser. The DACLs on the window station, desktop and current directory are then modified to allow the domain account access to them and the job is started in that window station and desktop with a call to CreateProcessAsUser. The starting directory is set to the same as sent by Crun, STDOUT and STDERR handles are set to the named pipes and the environment sent by Crun is used. CondorAuthSvc also starts a thread which waits on the new process handle until it terminates to close the named pipes. If the process starts correctly the process ID is returned to Crun.

If Crun requests that the job be stopped (again via RPC), CondorAuthSvc loops over all windows on the window station and desktop specified until it finds the one associated with the required process ID. It then sends that window a WM_CLOSE message, so any termination handling built in to the job should work correctly.

[Security Note: CondorAuthSvc currently makes no attempt to verify the origin of the call starting the job. This is, in principal, a bad thing since if the format of the RPC call is known it could let anyone start a job on the machine in the context of the domain user. If sensible security practices have been followed and the ACLs on sensitive system directories (such as C:\WINNT) do not allow write access to anyone other than trusted users the problem should not be too serious.]

Carun and AFSEncrypt
Carun and AFSEncrypt are a couple of utilities to allow jobs to access AFS

```
without any special recompilation. AFSEncrypt encrypts an AFS
username/password into a file (called .afs.xxx) using a simple XOR
algorithm. It's not a particularly secure way to do it, but it's simple and
self-inverse. Carun reads this file and gets an AFS token before running
whatever job is on its command line as a child process. It waits on the
process handle and a 24 hour timer. If the timer expires first it briefly
suspends the primary thread of the child process and attempts to get a new
AFS token before restarting the job, the idea being that the job should
have uninterrupted access to AFS if it runs for more than 25 hours (the
default token lifetime). As a security measure, the AFS credentials are
cached by Carun in memory and the .afs.xxx file deleted as soon as the
username/password have been read for the first time.

Carun needs the machine to be running either the IBM AFS client or the
OpenAFS client to work. It also needs the client libraries if you want to
rebuild it.

For example, if you wanted to get a list of your AFS tokens under Condor
you would run the following:

Crun \\mymachine\myshare\Carun tokens.exe

Running a job
To run a job using this mechanism specify the following in your job
submission (assuming Crun is in C:\CondorAuth):

Executable= c:\CondorAuth\Crun.exe
Arguments = \\mymachine\myshare\carun.exe
\\anothermachine\anothershare\myjob.exe
Transfer_Input_Files = .afs.xxx

along with your usual settings.

Installation
A basic installation script for use with the Inno Setup installation
package compiler can be found in the Install folder.
```

6.2.11 Interoperability between Condor for Unix and Condor for Windows

Unix machines and Windows machines running Condor can happily co-exist in the same Condor pool without any problems. Jobs submitted on Windows can run on Windows or Unix, and jobs submitted on Unix can run on Unix or Windows. Without any specification (using the `requirements` expression in the submit description file), the default behavior will be to require the execute machine to be of the same architecture and operating system as the submit machine.

There is absolutely no need to run more than one Condor central manager, even if you have both Unix and Windows machines. The Condor central manager itself can run on either Unix or Windows; there is no advantage to choosing one over the other. Here at University of Wisconsin-Madison, for instance, we have hundreds of Unix and Windows machines in our Computer Science Department Condor pool.

6.2.12 Some differences between Condor for Unix -vs- Condor for Windows

- On Unix, we recommend the creation of a "*condor*" account when installing Condor. On Windows, this is not necessary, as Condor is designed to run as a system service as user LocalSystem.
- On Unix, Condor finds the `condor_config` main configuration file by looking in ~condor, in /etc, or via an environment variable. On NT, the location of `condor_config` file is determined via the registry key `HKEY_LOCAL_MACHINE/Software/Condor`. You can override this value by setting an environment variable named `CONDOR_CONFIG`.
- On Unix, in the VANILLA universe at job vacate time Condor sends the job a softkill signal defined in the submit-description file (defaults to SIGTERM). On NT, Condor sends a WM_CLOSE message to the job at vacate time.
- On Unix, if one of the Condor daemons has a fault, a core file will be created in the `$(Log)` directory. On Condor NT, a "core" file will also be created, but instead of a memory dump of the process it will be a very short ASCII text file which describes what fault occurred and where it happened. This information can be used by the Condor developers to fix the problem.

6.3 Macintosh OS X

This section provides information specific to the Macintosh OS X port of Condor. The Macintosh port of Condor is more accurately a port of Condor to Darwin, the BSD core of OS X. Condor uses the Carbon library only to detect keyboard activity, and it does not use Cocoa at all. Condor on the Macintosh is a relatively new port, and it is not yet well-integrated into the Macintosh environment.

Condor on the Macintosh has a few shortcomings:

- Users connected to the Macintosh via *ssh* are not noticed for console activity.
- The memory size of threaded programs is reported incorrectly.
- No Macintosh-based installer is provided.
- The example start up scripts do not follow Macintosh conventions.
- Kerberos is not supported.

CHAPTER

SEVEN

Frequently Asked Questions (FAQ)

This is where you can find quick answers to some commonly asked questions about Condor.

7.1 Obtaining & Installing Condor

Where can I download Condor?

Condor can be downloaded from the mirrors listed at http://www.cs.wisc.edu/condor/downloads.

When I click to download Condor, it sends me back to the downloads page!

If you are trying to download Condor through a web proxy, try disabling it. Our web site uses the "referring page" as you navigate through our download menus in order to give you the right version of Condor, but sometimes proxies block this information from reaching our web site.

What platforms are supported?

Supported platforms are listed in section 1.5, on page 5. There is also platform-specific information at Chapter 6 on page 593.

Can I get the source code?

For Condor version 7.0.0 and later releases, the Condor source code is available for public download with the binary distributions.

What is Personal Condor?

Personal Condor is a term used to describe a specific style of Condor installation suited for individual users who do not have their own pool of machines, but want to submit Condor jobs to run elsewhere.

A Personal Condor is essentially a one-machine, self-contained Condor pool which can use *flocking* to access resources in other Condor pools. See Section 5.2, on page 553 for more information on flocking.

What do I do now? My installation of Condor does not work.

What to do to get Condor running properly depends on what sort of error occurs. One common error category are communication errors. Condor daemon log files report a failure to bind. For example:

```
(date and time) Failed to bind to command ReliSock
```

Or, the errors in the various log files may be of the form:

```
(date and time) Error sending update to collector(s)
(date and time) Can't send end_of_message
(date and time) Error sending UDP update to the collector

(date and time) failed to update central manager

(date and time) Can't send EOM to the collector
```

This problem can also be observed by running *condor_status*. It will give a message of the form:

```
Error:  Could not fetch ads --- error communication error
```

To solve this problem, understand that Condor uses the first network interface it sees on the machine. Since machines often have more than one interface, this problem usually implies that the wrong network interface is being used. It also may be the case that the system simply has the wrong IP address configured.

It is incorrect to use the localhost network interface. This has IP address 127.0.0.1 on all machines. To check if this incorrect IP address is being used, look at the contents of the CollectorLog file on the pool's your central manager right after it is started. The contents will be of the form:

```
5/25 15:39:33 ******************************************************
5/25 15:39:33 ** condor_collector (CONDOR_COLLECTOR) STARTING UP
5/25 15:39:33 ** $CondorVersion: 6.2.0 Mar 16 2001 $
5/25 15:39:33 ** $CondorPlatform: INTEL-LINUX-GLIBC21 $
5/25 15:39:33 ** PID = 18658
5/25 15:39:33 ******************************************************
5/25 15:39:33 DaemonCore: Command Socket at <128.105.101.15:9618>
```

The last line tells the IP address and port the collector has bound to and is listening on. If the IP address is 127.0.0.1, then Condor is definitely using the wrong network interface.

There are two solutions to this problem. One solution changes the order of the network interfaces. The preferred solution sets which network interface Condor should use by adding the following parameter to the local Condor configuration file:

```
NETWORK_INTERFACE = machine-ip-address
```

Where `machine-ip-address` is the IP address of the interface you wish Condor to use.

After an installation of Condor, why do the daemons refuse to start?

This message appears in the log files:

```
ERROR "The following configuration macros appear to contain default values
that must be changed before Condor will run.  These macros are:
hostallow_write
(found on line 1853 of /scratch/adesmet/TRUNK/work/src/localdir/condor_config)"
at line 217 in file condor_config.C
```

As of Condor 6.8.0, if Condor sees the bare key word: `YOU_MUST_CHANGE_THIS_INVALID_CONDOR_CONFIGURATION_VALUE` as the value of a configuration file entry, Condor daemons will log the given error message and exit.

By default, an installation of Condor 6.8.0 and later releases will have the configuration file entry `HOSTALLOW_WRITE` set to the above sentinel value. The Condor administrator must alter this value to be the correct domain or IP addresses that the administrator desires. The wild card character (`*`) may be used to define this entry, but that allows anyone, from anywhere, to submit jobs into the pool. A better value will be of the form `*.domainname.com`.

Why do standard universe jobs never run after an upgrade?

Standard universe jobs that remain in the job queue across an upgrade from any Condor release previous to 6.7.15 to any Condor release of 6.7.15 or more recent cannot run. They are missing a required ClassAd attribute (`LastCheckpointPlatform`) added for all standard universe jobs as of Condor version 6.7.15. This new attribute describes the platform where a job was running

when it produced a checkpoint. The attribute is utilized to identify platforms capable of continuing the job (using the checkpoint).

This attribute becomes necessary due to bugs in some Linux kernels. A standard universe job may be continued on some, but not all Linux machines. And, the `CkptOpSys` attribute is not specific enough to be utilized.

There are two possible solutions for these standard universe jobs that cannot run, yet are in the queue:

1. Remove and resubmit the standard universe jobs that remain in the queue across the upgrade. This includes all standard universe jobs that have flocked in to the pool. Note that the resubmitted jobs will start over again from the beginning.
2. For each standard universe job in the queue, modify its job ClassAd such that it can possibly run within the upgraded pool. If the job has already run and produced a checkpoint on a machine before the upgrade, determine the machine that produced the checkpoint using the `LastRemoteHost` attribute in the job's ClassAd. Then look at that machine's ClassAd (after the upgrade) to determine and extract the value of the `CheckpointPlatform` attribute. Add this (using *condor_qedit*) as the value of the new attribute `LastCheckpointPlatform` in the job's ClassAd. Note that this operation must also have to be performed on standard universe jobs flocking in to an upgraded pool. It is recommended that pools that flock between each other upgrade to a post 6.7.15 version of Condor.

Note that if the upgrade to Condor takes place at the same time as a platform change (such as booting an upgraded kernel), there is no way to properly set the `LastCheckpointPlatform` attribute. The only option is to remove and resubmit the standard universe jobs.

7.2 Setting up Condor

How do I set up a central manager on a machine with multiple network interfaces?

Please see section 3.7.3 on page 364.

How do I get more than one job to run on my SMP machine?

Condor will automatically recognize a SMP machine and advertise each CPU of the machine separately. For more details, see section 3.13.9 on page 443.

How do I configure a separate policy for the CPUs of an SMP machine?

Please see section 3.13.9 on page 443 for a lengthy discussion on this topic.

How do I set up my machines so that only specific users' jobs will run on them?

Restrictions on what jobs will run on a given resource are enforced by only starting jobs that meet specific constraints, and these constraints are specified as part of the configuration.

To specify that a given machine should only run certain users' jobs, and always run the jobs regardless of other activity on the machine, load average, etc., place the following entry in the machine's Condor configuration file:

```
START = ( (RemoteUser == "userfoo@baz.edu") || \
          (RemoteUser == "userbar@baz.edu") )
```

A more likely scenario is that the machine is restricted to run only specific users' jobs, contingent on the machine not having other interactive activity and not being heavily loaded. The following entries are in the machine's Condor configuration file. Note that extra configuration variables are defined to make the START variable easier to read.

```
# Only start jobs if:
# 1) the job is owned by the allowed users, AND
# 2) the keyboard has been idle long enough, AND
# 3) the load average is low enough OR the machine is currently
#    running a Condor job, and would therefore accept running
#    a different one
AllowedUser    = ( (RemoteUser == "userfoo@baz.edu") || \
                   (RemoteUser == "userbar@baz.edu") )
KeyboardUnused = (KeyboardIdle > $(StartIdleTime))
NoOwnerLoad    = ($(CPUIdle) || (State != "Unclaimed" && State != "Owner"))
START          = $(AllowedUser) && $(KeyboardUnused) && $(NoOwnerLoad)
```

To configure multiple machines to do so, create a common configuration file containing this entry for them to share.

How do I configure Condor to run my jobs only on machines that have the right packages installed?

This is a two-step process. First, you need to tell the machines to report that they have special software installed, and second, you need to tell the jobs to require machines that have that software.

To tell the machines to report the presence of special software, first add a parameter to their configuration files like so:

```
HAS_MY_SOFTWARE = True
```

And then, if there are already STARTD_ATTRS defined in that file, add HAS_MY_SOFTWARE to them, or, if not, add the line:

```
STARTD_ATTRS = HAS_MY_SOFTWARE, $(STARTD_ATTRS)
```

NOTE: For these changes to take effect, each *condor_startd* you update needs to be reconfigured with *condor_reconfig* -startd.

Next, to tell your jobs to only run on machines that have this software, add a requirements statement to their submit files like so:

```
Requirements = (HAS_MY_SOFTWARE =?= True)
```

NOTE: Be sure to use =?= instead of == so that if a machine doesn't have the HAS_MY_SOFTWARE parameter defined, the job's Requirements expression will not evaluate to "undefined", preventing it from running anywhere!

How do I configure Condor to only run jobs at night?

A commonly requested policy for running batch jobs is to only allow them to run at night, or at other pre-specified times of the day. Condor allows you to configure this policy with the use of the ClockMin and ClockDay *condor_startd* attributes. A complete example of how to use these attributes for this kind of policy is discussed in subsubsection 3.5.9 on page 308.

How do I configure Condor such that all machines do not produce checkpoints at the same time?

If machines are configured to produce checkpoints at fixed intervals, a large number of jobs are queued (submitted) at the same time, and these jobs start on machines at about the same time, then all these jobs will be trying to write out their checkpoints at the same time. It is likely to cause rather poor performance during this burst of writing.

The RANDOM_INTEGER() macro can help in this instance. Instead of defining PERIODIC_CHECKPOINT to be a fixed interval, each machine is configured to randomly choose one of a set of intervals. For example, to set a machine's interval for producing checkpoints to within the range of two to three hours, use the following configuration:

```
PERIODIC_CHECKPOINT = $(LastCkpt) > ( 2 * $(HOUR) + \
      $RANDOM_INTEGER(0,60,10) * $(MINUTE) )
```

The interval used is set at configuration time. Each machine is randomly assigned a different interval (2 hours, 2 hours and 10 minutes, 2 hours and 20 minutes, etc.) at which to produce checkpoints. Therefore, the various machines will not all attempt to produce checkpoints at the same time.

Why will the *condor_master* not run when a local configuration file is missing?

If a `LOCAL_CONFIG_FILE` is specified in the global configuration file, but the specified file does not exist, the *condor_master* will not start up, and it prints a variation of the following example message.

```
ERROR: Can't read config file /mnt/condor/hosts/bagel/condor_config.local
```

This is not a bug; it is a feature! Condor has always worked this way on purpose. There is a potentially large security hole if Condor is configured to read from a file that does not exist. By creating that file, a malicious user could change all sorts of Condor settings. This is an easy way to gain root access to a machine, where the daemons are running as root.

The intent is that if you've set up your global configuration file to read from a local configuration file, and the local file is not there, then something is wrong. It is better for the *condor_master* to exit right away and log an error message than to start up.

If the *condor_master* continued with the local configuration file missing, either A) someone could breach security or B) you will have potentially important configuration information missing. Consider the example where the local configuration file was on an NFS partition and the server was down. There would be all sorts of really important stuff in the local configuration file, and Condor might do bad things if it started without those settings.

If supplied it with an empty file, the *condor_master* works fine.

7.3 Running Condor Jobs

Why aren't any or all of my jobs running?

Please see Section 2.6.5, on page 43 for information on why a job might not be running.

I'm at the University of Wisconsin-Madison Computer Science Dept., and I am having problems!

Please see the web page http://www.cs.wisc.edu/condor/uwcs. As it explains, your home directory is in AFS, which by default has access control restrictions which can prevent Condor jobs from running properly. The above URL will explain how to solve the problem.

I'm getting a lot of e-mail from Condor. Can I just delete it all?

Generally you shouldn't ignore **all** of the mail Condor sends, but you can reduce the amount you get by telling Condor that you don't want to be notified every time a job successfully completes, only when a job experiences an error. To do this, include a line in your submit file like the following:

```
Notification = Error
```

See the Notification parameter in the *condor_q* man page on page 831 of this manual for more information.

Why will my vanilla jobs only run on the machine where I submitted them from?

Check the following:

1. Did you submit the job from a local file system that other computers can't access?
   See Section 3.3.7, on page 184.
2. Did you set a special requirements expression for vanilla jobs that's preventing them from running but not other jobs?
   See Section 3.3.7, on page 184.
3. Is Condor running as a non-root user?
   See Section 3.6.13, on page 350.

Why does the `requirements` expression for the job I submitted have extra things that I did not put in my submit description file?

There are several extensions to the submitted `requirements` that are automatically added by Condor. Here is a list:

- Condor automatically adds `arch` and `opsys` if not specified in the submit description file. It is assumed that the executable needs to execute on the same platform as the machine on which the job is submitted.
- Condor automatically adds the expression `(Memory * 1024 > ImageSize)`. This ensures that the job will run on a machine with at least as much physical memory as the memory footprint of the job.
- Condor automatically adds the expression `(Disk >= DiskUsage)` if not already specified. This ensures that the job will run on a machine with enough disk space for the job's local I/O (if there is any).

- A pool administrator may define configuration variables that cause expressions to be added to `requirements`. These configuration variables are `APPEND_REQUIREMENTS`, `APPEND_REQ_VANILLA`, and `APPEND_REQ_STANDARD`. These configuration variables give pool administrators the flexibility to set policy for a local pool.
- Older versions of Condor needed to add confusing clauses about WINNT and the FileSystemDomain to vanilla universe jobs. This made sure that the jobs ran on a machine where files were accessible. The Windows version supported automatically transferring files with the vanilla job, while the Unix version relied on a shared file system. Since the Unix version of Condor now supports transferring files, these expressions are no longer added to the `requirements` for a job.

When I use *condor_compile* to produce a job, I get an error that says, "Internal ld was not invoked!". What does this mean?

condor_compile enforces a specific behavior in the compilers and linkers that it supports (for example *gcc*, *g77*, *cc*, *CC*, *ld*) where a special linker script provided by Condor must be invoked during the final linking stages of the supplied compiler or linker.

In some rare cases, as with *gcc* compiled with the options **–with-as** or **–with-ld**, the enforcement mechanism we rely upon to have *gcc* choose our supplied linker script is not honored by the compiler. When this happens, an executable is produced, but the executable is devoid of the Condor libraries which both identify it as a Condor executable linked for the standard universe and implement the feature sets of remote I/O and transparent process checkpointing and migration.

Often, the only fix in order to use the compiler desired, is to reconfigure and recompile the compiler itself, such that it does not use the errant options mentioned.

With Condor's standard universe, we highly recommend that your source files are compiled with the supported compiler for your platform. See section 1.5 for the list of supported compilers. For a Linux platform, the supported compiler is the default compiler that came with the distribution. It is often found in the directory `/usr/bin`.

Why might my job be preempted (evicted)?

There are four circumstances under which Condor may evict a job. They are controlled by different expressions.

Reason number 1 is the user priority: controlled by the `PREEMPTION_REQUIREMENTS` expression in the configuration file. If there is a job from a higher priority user sitting idle, the *condor_negotiator* daemon may evict a currently running job submitted from a lower priority user if `PREEMPTION_REQUIREMENTS` is True. For more on user priorities, see section 2.7 and section 3.4.

Reason number 2 is the owner (machine) policy: controlled by the `PREEMPT` expression in the

configuration file. When a job is running and the PREEMPT expression evaluates to True, the *condor_startd* will evict the job. The PREEMPT expression should reflect the requirements under which the machine owner will not permit a job to continue to run. For example, a policy to evict a currently running job when a key is hit or when it is the 9:00am work arrival time, would be expressed in the PREEMPT expression and enforced by the *condor_startd*. For more on the PREEMPT expression, see section 3.5.

Reason number 3 is the owner (machine) preference: controlled by the RANK expression in the configuration file (sometimes called the startd rank or machine rank). The RANK expression is evaluated as a floating point number. When one job is running, a second idle job that evaluates to a higher RANK value tells the *condor_startd* to prefer the second job over the first. Therefore, the *condor_startd* will evict the first job so that it can start running the second (preferred) job. For more on RANK, see section 3.5.

Reason number 4 is if Condor is to be shutdown: on a machine that is currently running a job. Condor evicts the currently running job before proceeding with the shutdown.

Condor does not stop the Condor jobs running on my Linux machine when I use my keyboard and mouse. Is there a bug?

There is no bug in Condor. Unfortunately, recent Linux 2.4.x and all Linux 2.6.x kernels through version 2.6.10 do not post proper state information, such that Condor can detect keyboard and mouse activity. Condor implements workarounds to piece together the needed state information for PS/2 devices. A better fix of the problem utilizes the kernel patch linked to from the directions posted at http://www.cs.wisc.edu/condor/kernel.patch.html. This patch works better for PS/2 devices, and may also work for USB devices. A future version of Condor will implement better recognition of USB devices, such that the kernel patch will also definitively work for USB devices.

What signals get sent to my jobs when Condor needs to preempt or kill them, or when I remove them from the queue? Can I tell Condor which signals to send?

The answer is dependent on the universe of the jobs.

Under the scheduler universe, the signal jobs get upon *condor_rm* can be set by the user in the submit description file with the form of

```
remove_kill_sig = SIGWHATEVER
```

If this command is not defined, Condor further looks for a command in the submit description file with the form

```
kill_sig = SIGWHATEVER
```

And, if that command is also not given, Condor uses SIGTERM.

For all other universes, the jobs get the value of the submit description file command `kill_sig`, which is SIGTERM by default.

If a job is killed or evicted, the job is sent a `kill_sig`, unless it is on the receiving end of a hard kill, in which case it gets SIGKILL.

Under all universes, the signal is sent only to the parent PID of the job, namely, the first child of the *condor_starter*. If the child itself is forking, the child must catch and forward signals as appropriate. This in turn depends on the user's desired behavior. The exception to this is (again) where the job is receiving a hard kill. Condor sends the value SIGKILL to all the PIDs in the family.

Why does my Linux job have an enormous ImageSize and refuse to run anymore?

Sometimes Linux jobs run, are preempted and can not start again because Condor thinks the image size of the job is too big. This is because Condor has a problem calculating the image size of a program on Linux that uses threads. It is particularly noticeable in the Java universe, but it also happens in the vanilla universe. It is not an issue in the standard universe, because threaded programs are not allowed.

On Linux, each thread appears to consume as much memory as the entire program consumes, so the image size appears to be (number-of-threads * image-size-of-program). If your program uses a lot of threads, your apparent image size balloons. You can see the image size that Condor believes your program has by using the -l option to condor_q, and looking at the ImageSize attribute.

When you submit your job, Condor creates or extends the requirements for your job. In particular, it adds a requirement that you job must run on a machine with sufficient memory:

```
Requirements = ... ((Memory * 1024) >= ImageSize) ...
```

(Note that memory is the execution machine's memory in megabytes while ImageSize is in kilobytes). When your application is threaded, the image size appears to be much larger than it really is, and you may not have a machine with sufficient memory to handle this requirement.

Unfortunately, calculating the correct ImageSize is rather hard to fix on Linux, and we do not yet have a good solution. Fortunately, there is a workaround while we work on a good solution for a future release.

In the Requirements expression above, Condor added `(Memory * 1024) >= ImageSize)` on your behalf. You can prevent Condor from doing this by giving it your own expression about memory in your submit file, just as:

```
Requirements = Memory > 1024
```

You will need to change 1024 to a reasonably good estimate of the actual image size of your program, in kilobytes. This expression says that your program requires 1 megabyte of memory. If you underestimate the memory your application needs, you may have bad performance if you job runs on machines that have insufficient memory.

In addition, if you have modified your machine policies to preempt jobs when they get big a ImageSize, you will need to change those policies.

Why does the time output from *condor_status* appear as [?????] ?

Condor collects timing information for a large variety of uses. Collection of the data relies on accurate times. Being a distributed system, clock skew among machines causes errant timing calculations. Values can be reported too large or too small, with the possibility of calculating negative timing values.

This problem may be seen by the user when looking at the output of *condor_status*. If the **ActivityTime** field appears as [?????], then this calculated statistic was negative. *condor_status* recognizes that a negative amount of time will be nonsense to report, and instead displays this string.

The solution to the problem is to synchronize the clocks on these machines. An administrator can do this using a tool such as *ntp*.

The user condor's home directory cannot be found. Why?

This problem may be observed after installation, when attempting to execute

```
~condor/condor/bin/condor_config_val  -tilde
```

and there is a user named condor. The command prints a message such as

```
Error: Specified -tilde but can't find condor's home directory
```

In this case, the difficulty stems from using NIS, because the Condor daemons fail to communicate properly with NIS to get account information. To fix the problem, a dynamically linked version of Condor must be installed.

Condor commands (including *condor_q*) are really slow. What is going on?

Some Condor programs will react slowly if they expect to find a *condor_collector* daemon, yet cannot contact one. Notably, *condor_q* can be very slow. The *condor_schedd* daemon will also be slow, and it will log lots of harmless messages complaining. If you are not running a *condor_collector* daemon, it is important that the configuration variable `COLLECTOR_HOST` be set to nothing. This is typically done by setting `CONDOR_HOST` with

```
CONDOR_HOST=
COLLECTOR_HOST=$(CONDOR_HOST)
```

or

```
COLLECTOR_HOST=
```

Where are my missing files? The command `when_to_transfer_output = ON_EXIT_OR_EVICT` is in the submit description file.

Although it may appear as if files are missing, they are not. The transfer does take place whenever a job is preempted by another job, vacates the machine, or is killed. Look for the files in the directory defined by the SPOOL configuration variable. See section 2.5.4, on page 25 for details on the naming of the intermediate files.

Why are my vm universe VMware jobs failing and being put on hold?

Strange behavior has been noted when Condor tries to run a **vm** universe VMware job using a path to a VMX file that contains a symbolic link. An example of an error message that may appear in such a job's user log:

```
Error from starter on master_vmuniverse_strtd@nostos.cs.wisc
.edu: register(/scratch/gquinn/condor/git/CONDOR_SRC/src/con
dor_tests/31426/31426vmuniverse/execute/dir_31534/vmN3hylp_c
ondor.vmx) = 1/Error: Command failed: A file was not found/(
ERROR) Can't create snapshot for vm(/scratch/gquinn/condor/g
it/CONDOR_SRC/src/condor_tests/31426/31426vmuniverse/execute
/dir_31534/vmN3hylp_condor.vmx)
```

To work around this problem:

- If using file transfer (the submit description file contains **vmware_should_transfer_files** = **true**), then modify any configuration variable EXECUTE values on all execute machines, such that they do not contain symbolic link path components.
- If using a shared file system, ensure that the submit description file command **vmware_dir** does not use symbolic link path name components.

7.4 Condor on Windows

Will Condor work on a network of mixed Unix and Windows machines?

You can have a Condor pool that consists of both Unix and Windows machines.

Your central manager can be either Windows or Unix. For example, even if you had a pool consisting strictly of Unix machines, you could use a Windows box for your central manager, and vice versa.

Submitted jobs can originate from either a Windows **or** a Unix machine, and be destined to run on Windows **or** a Unix machine. Note that there are still restrictions on the supported universes for jobs executed on Windows machines.

So, in summary:

1. A single Condor pool can consist of both Windows and Unix machines.
2. It does not matter at all if your Central Manager is Unix or Windows.
3. Unix machines can submit jobs to run on other Unix or Windows machines.
4. Windows machines can submit jobs to run on other Windows or Unix machines.

What versions of Windows will Condor run on?

See Section 1.5, on page 5.

My Windows program works fine when executed on its own, but it does not work when submitted to Condor.

First, make sure that the program really does work outside of Condor under Windows, that the disk is not full, and that the system is not out of user resources.

As the next consideration, know that some Windows programs do not run properly because they are dynamically linked, and they cannot find the `.dll` files that they depend on. Version 6.4.x of Condor sets the `PATH` to be empty when running a job. To avoid these difficulties, do one of the following

1. statically link the application
2. wrap the job in a script that sets up the environment
3. submit the job from a correctly-set environment with the command

```
getenv = true
```

in the submit description file. This will copy your environment into the job's environment.

4. send the required `.dll` files along with the job using the submit description file command **transfer_input_files**.

Why is the *condor_master* daemon failing to start, giving an error about "In StartServiceCtrlDispatcher, Error number: 1063"?

In Condor for Windows, the *condor_master* daemon is started as a service. Therefore, starting the *condor_master* daemon as you would on Unix will not work. Start Condor on Windows machines using either

```
net start condor
```

or start the Condor service from the Service Control Manager located in the Windows Control Panel.

Jobs submitted from Windows give an error referring to a credential.

Jobs submitted from a Windows machine require a stashed password in order for Condor to perform certain operations on the user's behalf. Refer to section 6.2.3 for information about password storage on Windows. The command which stashes a password for a user is *condor_store_cred*. See the manual page on on page 823 for usage details.

The error message that Condor gives if a user has not stashed a password is of the form:

```
ERROR: No credential stored for username@machinename

        Correct this by running:
        condor_store_cred add
```

Jobs submitted from Unix to execute on Windows do not work properly.

A difficulty with defaults causes jobs submitted from Unix for execution on a Windows platform to remain in the queue, but make no progress. For jobs with this problem, log files will contain error messages pointing to shadow exceptions.

This difficulty stems from the defaults for whether file transfer takes place. The workaround for this problem is to place the line

```
TRANSFER_FILES  = ALWAYS
```

into the submit description file for jobs submitted from a Unix machine for execution on a Windows machine.

When I run *condor_status* I get a communication error, or the Condor daemon log files report a failure to bind.

Condor uses the first network interface it sees on your machine. This problem usually means you have an extra, inactive network interface (such as a RAS dial up interface) defined before the regular network interface.

To solve this problem, either change the order of the network interfaces in the Control Panel, or explicitly set which network interface Condor should use by adding the following definition to the Condor configuration file:

```
NETWORK_INTERFACE = <ip-address>
```

Where `<ip-address>` is the IP address of the interface that Condor is to use.

My job starts but exits right away with status 128.

This can occur when the machine your job is running on is missing a DLL (Dynamically Linked Library) required by your program. The solution is to find the DLL file the program needs and put it in the TRANSFER_INPUT_FILES list in the job's submit file.

To find out what DLLs your program depends on, right-click the program in Explorer, choose Quickview, and look under "Import List".

How can I access network files with Condor on Windows?

Five methods for making access of network files work with Condor are given in section 6.2.10.

What is wrong when *condor_off* cannot find my host, and *condor_status* does not give me a complete host name?

Given the command

```
condor_off hostname2
```

an error message of the form

```
Can't find address for master hostname2.somewhere.edu
```

appears. Yet, when looking at the host names with

```
condor_status -master
```

the output is of the form

```
hostname1.somewhere.edu
hostname2
hostname3.somewhere.edu
```

To correct this incomplete host name, add an entry to the configuration file for DEFAULT_DOMAIN_NAME that specifies the domain name to be used. For the example given, the configuration entry will be

```
DEFAULT_DOMAIN_NAME = somewhere.edu
```

After adding this configuration file entry, use *condor_restart* to restart the Condor daemons and effect the change.

Does USER_JOB_WRAPPER work on Windows machines?

The USER_JOB_WRAPPER configuration variable does work on Windows machines. The wrapper must be either a batch script with a file name extension of .bat or .cmd, or an executable with a file name extension of .exe or .com.

An example of a batch script sets environment variables:

```
REM set some environment variables
set LICENSE_SERVER=192.168.1.202:5012
set MY_PARAMS=2

REM Run the actual job now
%*
```

condor_store_cred is failing, and I'm sure I'm typing my password correctly.

First, make sure the *condor_schedd* daemon is running.

Next, check the log file written by the *condor_schedd* daemon. It will contain more detailed information about the failure. Frequently, the error is a result of PERMISSION DENIED errors. More information about proper configuration of security settings is on page 342.

My submit machine cannot have more than 120 jobs running concurrently. Why?

Windows is likely to be running out of desktop heap. Confirm this to be the case by looking in the log for the *condor_schedd* daemon to see if *condor_shadow* daemons are immediately exiting with status 128. If this is the case, increase the desktop heap size. Open the registry key:

```
HKEY_LOCAL_MACHINE\System\CurrentControlSet\Control\Session Manager\SubSystems\Windows
```

The SharedSection value can have three values separated by commas. The third value controls the desktop heap size for non-interactive desktops, which the Condor service uses. The default is 512 (Kbytes). 60 *condor_shadow* daemons consume about 256 Kbytes, hence 120 shadows can run with the default value. To be able to run a maximum of 300 *condor_shadow* daemons, set this value at 1280.

Reboot the system for the changes to take effect. For more information, see Microsoft Article Q184802.

Why do Condor daemons exit after logging a 10038 (WSAENOTSOCK) error on some machines?

Usually when Condor daemons exit in this manner, it is because the system in question has a non-standard Winsock Layered Service Provider (LSP) installed on it. An LSP is, in effect, a plug-in for the TCP/IP protocol stack. LSPs have been installed as part of anti-virus software and other security-related packages.

There are several tools available to check your system for the presence of LSPs. One with which we have had success is *LSP-Fix*, available at http://www.cexx.org/lspfix.htm. Any non-Microsoft LSPs identified by this tool may potentially be causing the WSAENOTSOCK error in Condor. Although the *LSP-Fix* tool allows the direct removal of an LSP, it is likely advisable to completely remove the application for which the LSP is a part via the Control Panel.

Another approach is to completely reset the TCP/IP stack to its original state. This can be done using the `netsh` tool:

```
netsh int ip reset reset-stack.log
```

The command will return the TCP/IP stack back to the state is was in when the OS was first installed. The log file defined above will record all the configuration changes made by `netsh`.

Why do Condor daemons exit with "Unexpected performance counter size", "unable to spawn the ProcD" or "loadavg thread died, restarting. (exit code=2)" errors?

Condor on Windows platforms relies on built-in performance counters for its operation. If performance counters that Condor requires are disabled, daemons may exit with a message such as

```
1/26 09:16:42 (fd:2) (pid:5732) ERROR: "Unexpected performance counter
    size for total CPU: 0 (expected: 8)" at line 2846 in file
    ..\src\condor_procapi\procapi.cpp
```

or

```
1/20 15:29:14 (pid:2484) ERROR "unable to spawn the ProcD" at line 136
    in file ..\src\condor_c++_util\proc_family_proxy.C
```

and even

```
4/16 10:49:13 loadavg thread died, restarting. (exit code=2)
```

To enable the performance counters, check the registry key

```
HKEY_LOCAL_MACHINE\SYSTEM\CurrentControlSet\Services\PerfProc\Performance
```

If a value for `Disable Performance Counters` exists, delete it or set it to `0`.

Why does the Windows Installer fail with “Error 2738. Could not access VBScript run time for custom action”?

This error results when the VBScript engine is not registered. Since Condor’s installer depends on the VBScript engine for custom steps, the installer will fail if it cannot find the VBScript engine.

The fix is to register the VMScript engine. With Administrative privilege:

1. Launch the Command Prompt (*cmd.exe*) as the Administrator.
2. At the Command Prompt, change directories to the `System32` folder, within the Windows folder.
3. Issue the command

   ```
   regsvr32 vbscript.dll
   ```

If successful, the message

```
DllRegisterServer in vbscript.dll succeeded.
```

is printed.

Why does Condor sometimes fail to parse floating point numbers?

Condor assumes that all floating point numbers are of the form x.y, which, depending on a computer's current locale, may not always be the case. This problem occurs even if Condor is running under an account that has had the locale configured correctly. The problem lies in the template user account which is used to create Condor's dynamic accounts. Even if the entire system is configured to use a new locale, this template account seems to retain the original system locale. The following steps can be used fix this problem.

To create a default user profile, you must be logged on as **Administrator** or be a member of the **Administrators** group. Create a new user profile for all new user accounts on a computer to be based on. To create subsequent profiles, you can use the new user account as a template. Here is how to use the new user profile as a template account to use as a new user's profile:

1. **Log on** to the computer as the new user, and customize the desktop if appropriate.
2. Optionally, install and configure any applications to be shared by user accounts made from this template.
3. **Log off**, and then log on as the **Administrator**.
4. In the **Control Panel**, open the **System** Control Panel applet.
    - On **Vista** click on the **Advanced system settings Task** listed in the left pane.
5. On the **Advanced** tab, under **User Profiles**, click **Settings**.
6. Under **Profiles stored on this computer**, select the user you created to be the template, and then click **Copy To**.
7. To create the default user profile for the computer, type the path to the default user:
    - On Windows 2000: `%WinDir%\Profiles\Default`;
    - On Windows XP: `%SystemDrive%\Documents and Settings\Defualt`;
    - On Vista: `%SystemDrive%\Users\Default`.
8. In the **Copy To** dialog box, under **Permitted to use**, click **Change**.
9. In the **Select User or Group** dialog box, in the **Enter the object name to select** text box, type: **Everyone** and click **OK**.
10. Click **OK** to dismiss the **Copy To** dialog box.

11. Click **OK** again to dismiss the **User Profiles** dialog box.
12. Finally, click **OK** one last time to dismiss the **System Properties** dialog.

If Condor has already created some dynamic accounts, you will need to remove them so that Condor can re-create them with the new template account.

7.5 Grid Computing

What must be installed to access grid resources?

A single machine with Condor installed such that jobs may be submitted is the minimum software necessary. If matchmaking or glidein is desired, then a single machine must not only be running Condor such that jobs may be submitted, but also fill the role of a central manager. A Personal Condor installation may satisfy both.

I am the administrator at Physics, and I have a 64-node cluster running Condor. The administrator at Chemistry is also running Condor on her 64-node cluster. We would like to be able to share resources. How do we do this?

Condor's flocking feature allows multiple Condor pools to share resources. By setting configuration variables within each pool, jobs may be executed on either cluster. See the manual section on flocking, section 5.2, for details.

What is glidein?

Glidein provides a way to temporarily add a resource to a local Condor pool. Glidein uses Globus resource-management software to run jobs on the resource. Those jobs are initially portions of Condor software, such that Condor is running on the resource, configured to be part of the local pool. Then, Condor may execute the user's jobs. There are several benefits to working in this way. Standard universe jobs may be submitted to run on the resource. Condor can also dynamically schedule jobs across the grid.

See the section on Glidein, section 5.4 of the manual for further information.

Using my Globus gatekeeper to submit jobs to the Condor pool does not work. What is wrong?

The Condor configuration file is in a non-standard location, and the Globus software does not know how to locate it, when you see either of the following error messages.

first error message

```
% globus-job-run \
  globus-gate-keeper.example.com/jobmanager-condor /bin/date

Neither the environment variable CONDOR_CONFIG, /etc/condor/,
nor ~condor/ contain a condor_config file.  Either set
CONDOR_CONFIG to point to a valid config file, or put a
"condor_config" file in /etc/condor or ~condor/ Exiting.

GRAM Job failed because the job failed when the job manager
attempted to run it (error code 17)
```

second error message

```
% globus-job-run \
   globus-gate-keeper.example.com/jobmanager-condor /bin/date

ERROR: Can't find address of local schedd GRAM Job failed
because the job failed when the job manager attempted to run it
(error code 17)
```

As described in section 3.2.2, Condor searches for its configuration file using the following ordering.

1. File specified in the `CONDOR_CONFIG` environment variable
2. `/etc/condor/condor_config`
3. `~condor/condor_config`
4. `$(GLOBUS_LOCATION)/etc/condor_config`

Presuming the configuration file is not in a standard location, you will need to set the `CONDOR_CONFIG` environment variable by hand, or set it in an initialization script. One of the following solutions for an initialization may be used.

1. Wherever *globus-gatekeeper* is launched, replace it with a minimal shell script that sets `CONDOR_CONFIG` and then starts *globus-gatekeeper*. Something like the following should work:

   ```
   #! /bin/sh
   CONDOR_CONFIG=/path/to/condor_config
   export CONDOR_CONFIG
   exec /path/to/globus/sbin/globus-gatekeeper "$@"
   ```

2. If you are starting *globus-gatekeeper* using *inetd*, *xinetd*, or a similar program, set the environment variable there. If you are using *inetd*, you can use the *env* program to set the environment. This example does this; the example is shown on multiple lines, but it will be all on one line in the *inetd* configuration.

```
globus-gatekeeper stream tcp nowait root /usr/bin/env
env CONDOR_CONFIG=/path/to/condor_config
/path/to/globus/sbin/globus-gatekeeper
-co /path/to/globus/etc/globus-gatekeeper.conf
```

If you're using *xinetd*, add an env setting something like the following:

```
service gsigatekeeper
{
    env = CONDOR_CONFIG=/path/to/condor_config
    cps = 1000 1
    disable = no
    instances = UNLIMITED
    max_load = 300
    nice = 10
    protocol = tcp
    server = /path/to/globus/sbin/globus-gatekeeper
    server_args = -conf /path/to/globus/etc/globus-gatekeeper.conf
    socket_type = stream
    user = root
    wait = no
}
```

7.6 Managing Large Workflows

How do I get meaningful output from *condor_q* with so many jobs in the queue?

There are several ways to constrain the output of *condor_q* when there are lots and lots of jobs in the queue. To see only the jobs that are currently running:

```
condor_q -run
```

To see only the jobs that are currently on hold:

```
condor_q -hold
```

To see other output, combine options. For example, to see only running jobs submitted by A. Einstein that belong to cluster 77:

```
condor_q -run einstein 77
```

Another example uses the **-constraint** option to *condor_q*. To see only the jobs in the queue that started running, but were interrupted and then started again from the beginning, perhaps more than once:

```
condor_q -constraint 'NumJobStarts > 1'
```

Complete details of *condor_q* are contained in the manual page at page 772.

What does Condor offer that can help with running a large number of jobs?

Many of the features of DAGMan are targeted at helping with the administration and running of large numbers of jobs. See section 2.10.12 at page 101.

7.7 Troubleshooting

If I see `PERMISSION DENIED` in my log files, what does that mean?

Most likely, the Condor installation has been misconfigured and Condor's access control security functionality is preventing daemons and tools from communicating with each other. Other symptoms of this problem include Condor tools (such as *condor_status* and *condor_q*) not producing any output, or commands that appear to have no effect (for example, *condor_off* or *condor_on*).

The solution is to properly configure the `HOSTALLOW_*` and `HOSTDENY_*` settings (for host/IP based authentication) or to configure strong authentication and set `ALLOW_*` and `DENY_*` as appropriate. Host-based authentication is described in section 3.6.9 on page 342. Information about other forms of authentication is provided in section 3.6.1 on page 315.

What happens if the central manager crashes?

If the central manager crashes, jobs that are already running will continue to run unaffected. Queued jobs will remain in the queue unharmed, but can not begin running until the central manager is restarted and begins matchmaking again. Nothing special needs to be done after the central manager is brought back on line.

Why did the *condor_schedd* daemon die and restart?

The *condor_schedd* daemon receives signal 25, dies, and is restarted when the history file reaches a 2 Gbyte size limit. Until a larger history file size or the rotation of the history file is supported in Condor, try one of these work arounds:

1. When the history file becomes large, remove it. Note that this causes a loss of the information in the history file, but the *condor_schedd* daemon will not die.
2. When the history file becomes large, move it.
3. Stop keeping the history. Only *condor_history* accesses the history file, so this particular functionality will be gone. To stop keeping the history, place

   ```
   HISTORY=
   ```

in the configuration, followed by a *condor_reconfig* command to recognize the change in currently executing daemons.

When I ssh/telnet to a machine to check particulars of how Condor is doing something, it is always vacating or unclaimed when I know a job had been running there!

Depending on how your policy is set up, Condor will track *any* tty on the machine for the purpose of determining if a job is to be vacated or suspended on the machine. It could be the case that after you ssh there, Condor notices activity on the tty allocated to your connection and then vacates the job.

What is wrong? I get no output from *condor_status*, but the Condor daemons are running.

One likely error message within the collector log of the form

```
DaemonCore: PERMISSION DENIED to host <xxx.xxx.xxx.xxx> for command 0 (UPDATE_STARTD_AD)
```

indicates a permissions problem. The *condor_startd* daemons do not have write permission to the *condor_collector* daemon. This could be because you used domain names in your HOSTALLOW_WRITE and/or HOSTDENY_WRITE configuration macros, but the domain name server (DNS) is not properly configured at your site. Without the proper configuration, Condor cannot resolve the IP addresses of your machines into fully-qualified domain names (an inverse look up). If this is the problem, then the solution takes one of two forms:

1. Fix the DNS so that inverse look ups (trying to get the domain name from an IP address) works for your machines. You can either fix the DNS itself, or use the DEFAULT_DOMAIN_NAME setting in your Condor configuration file.
2. Use numeric IP addresses in the HOSTALLOW_WRITE and/or HOSTDENY_WRITE configuration macros instead of domain names. As an example of this, assume your site has a machine such as foo.your.domain.com, and it has two subnets, with IP addresses 129.131.133.10, and 129.131.132.10. If the configuration macro is set as

   ```
   HOSTALLOW_WRITE = *.your.domain.com
   ```

   and this does not work, use

   ```
   HOSTALLOW_WRITE = 192.131.133.*, 192.131.132.*
   ```

Alternatively, this permissions problem may be caused by being too restrictive in the setting of your HOSTALLOW_WRITE and/or HOSTDENY_WRITE configuration macros. If it is, then the solution is to change the macros, for example from

```
HOSTALLOW_WRITE = condor.your.domain.com
```

to

```
HOSTALLOW_WRITE = *.your.domain.com
```

or possibly

```
HOSTALLOW_WRITE = condor.your.domain.com, foo.your.domain.com, \
bar.your.domain.com
```

Another likely error message within the collector log of the form

```
DaemonCore: PERMISSION DENIED to host <xxx.xxx.xxx.xxx> for command 5 (QUERY_STARTD_ADS)
```

indicates a similar problem as above, but read permission is the problem (as opposed to write permission). Use the solutions given above.

Why does Condor leave mail processes around?

Under FreeBSD and Mac OSX operating systems, misconfiguration of of a system's outgoing mail causes Condor to inadvertently leave paused and zombie mail processes around when Condor attempts to send notification e-mail. The solution to this problem is to correct the mailer configuration.

Execute the following command as the user under which Condor daemons run to determine whether outgoing e-mail works.

```
$ uname -a | mail -v your@emailaddress.com
```

If no e-mail arrives, then outgoing e-mail does not work correctly.

Note that this problem does not manifest itself on non-BSD Unix platforms, such as Linux.

Why are there spurious Condor errors on some machines running Xen kernels?

Some older Xen kernels had a problem where the kernel's jiffy counter could jump backwards in time. This breaks an assumption made by the *condor_procd*. This problem can only be worked around by upgrading the Xen kernel to a version that fixes the issue with the jiffy counter. Running Condor on an affected Xen kernel often results in failures of the following forms in Condor daemon log files:

- `error: parent process's birthday is later than our own`
- `ERROR: No family with the given PID is registered`

7.8 Other questions

Is there a Condor mailing-list?

Yes. There are two useful mailing lists. First, we run an extremely low traffic mailing list solely to announce new versions of Condor. Follow the instructions for Condor World at http://www.cs.wisc.edu/condor/mail-lists/. Second, our users can be extremely knowledgeable, and they help each other solve problems using the Condor Users mailing list. Again, follow the instructions for Condor Users at http://www.cs.wisc.edu/condor/mail-lists/.

My question isn't in the FAQ!

If you have any questions that are not listed in this FAQ, try looking through the rest of the manual. Try joining the Condor Users mailing list, where our users support each other in finding answers to problems. Follow the instructions at http://www.cs.wisc.edu/condor/mail-lists/. If you still can't find an answer, feel free to contact us at condor-admin@cs.wisc.edu.

Note that Condor's free e-mail support is provided on a best-effort basis, and at times we may not be able to provide a timely response. If guaranteed support is important to you, please inquire about our paid support services.

CHAPTER

EIGHT

Version History and Release Notes

8.1 Introduction to Condor Versions

This chapter provides descriptions of what features have been added or bugs fixed for each version of Condor. The first section describes the Condor version numbering scheme, what the numbers mean, and what the different *release series* are. The rest of the sections each describe a specific release series, and all the Condor versions found in that series.

8.1.1 Condor Version Number Scheme

Starting with version 6.0.1, Condor adopted a new, hopefully easy to understand version numbering scheme. It reflects the fact that Condor is both a production system and a research project. The numbering scheme was primarily taken from the Linux kernel's version numbering, so if you are familiar with that, it should seem quite natural.

There will usually be two Condor versions available at any given time, the *stable* version, and the *development* version. Gone are the days of "patch level 3", "beta2", or any other random words in the version string. All versions of Condor now have exactly three numbers, seperated by "."

- The first number represents the major version number, and will change very infrequently.
- *The thing that determines whether a version of Condor is* stable *or* development *is the second digit. Even numbers represent stable versions, while odd numbers represent development versions.*
- The final digit represents the minor version number, which defines a particular version in a given release series.

8.1.2 The Stable Release Series

People expecting the stable, production Condor system should download the stable version, denoted with an even number in the second digit of the version string. Most people are encouraged to use this version. We will only offer our paid support for versions of Condor from the stable release series.

On the stable series, new minor version releases will only be made for bug fixes and to support new platforms. No new features will be added to the stable series. People are encouraged to install new stable versions of Condor when they appear, since they probably fix bugs you care about. Hopefully, there will not be many minor version releases for any given stable series.

8.1.3 The Development Release Series

Only people who are interested in the latest research, new features that haven't been fully tested, etc, should download the development version, denoted with an odd number in the second digit of the version string. We will make a best effort to ensure that the development series will work, but we make no guarantees.

On the development series, new minor version releases will probably happen frequently. People should not feel compelled to install new minor versions unless they know they want features or bug fixes from the newer development version.

Most sites will probably never want to install a development version of Condor for any reason. Only if you know what you are doing (and like pain), or were explicitly instructed to do so by someone on the Condor Team, should you install a development version at your site.

After the feature set of the development series is satisfactory to the Condor Team, we will put a code freeze in place, and from that point forward, only bug fixes will be made to that development series. When we have fully tested this version, we will release a new stable series, resetting the minor version number, and start work on a new development release from there.

8.2 Upgrading from the 7.4 series to the 7.6 series of Condor

Upgrading from the 7.4 series of Condor to this 7.6 series will bring many features introduced in the 7.5 series of Condor. While nothing will replace reading through the version histories, here is a list of some new features and release notes to be aware of.

- Condor's RPMs now use FHS locations for files. See section 3.2.6 for updated installation information.
- The locations of *many* executables within the release directories have changed.
- Condor's file transfer mechanism now provided limited support for the transfer of directories.

- The feature set once known as the Startd Cron or as Hawkeye is now called *Daemon ClassAd Hooks*. Besides the new name, the mechanisms have been updated and significantly revised. See section 4.4.3 for documentation.
- Condor DAGMan provides new features. See section 2.10 for documentation.
- Grid jobs targeted at CREAM, EC2 servers and Eucalyptus systems are now supported.
- For those who compile Condor from the source code, rather than using packages of pre-built executables, Condor is now built using *cmake* instead of *imake*.
- Quill is now available in the source code, and it is not incorporated as part of pre-built and released executables.
- The *condor_job_router* is no longer limited to routing vanilla universe jobs.
- Condor uses the New ClassAd language internally.
- The *condor_shared_port* daemon allows Condor daemons to share a single network port. This makes opening access to Condor through a firewall easier and safer. It also increases the scalability of a submit node by decreasing port usage.

8.3 Stable Release Series 7.6

This is a stable release series of Condor. As usual, only bug fixes (and potentially, ports to new platforms) will be provided in future 7.6.x releases. New features will be added in the 7.7.x development series.

The details of each version are described below.

Version 7.6.0

Release Notes:

- Condor version 7.6.0 not yet released.
- Prior to Condor version 7.5.0, commenting out PREEN in the default configuration file disabled *condor_preen*. Starting in Condor version 7.5.0, there was an internal default value for PREEN, so if the configuration variable was not set in any configuration file, *condor_preen* would still run. To disable this functionality, PREEN can be explicitly set to nothing.

New Features:

- Condor can now create and manage virtual machine instances in a cloud service via Deltacloud. This is done via the new **deltacloud** grid type in the grid universe. See section 5.3.9 for details.

- Improved scalability of submission of cream grid type jobs.

Configuration Variable and ClassAd Attribute Additions and Changes:

- The new configuration variable `DELTACLOUD_GAHP` specifies where the *deltacloud_gahp* binary is located. This binary is used to manage deltacloud grid type jobs in the grid universe. In a normal Condor installation, the value should be `$(SBIN)/deltacloud_gahp`.
- Several new job ClassAd attributes have been added to support the deltacloud grid type in the grid universe. These attributes are taken from the submit description file: `DeltacloudUsername`, `DeltacloudPasswordFile`, `DeltacloudImageId`, `DeltacloudRealmId`, `DeltacloudHardwareProfile`, `DeltacloudHardwareProfileCpu`, `DeltacloudHardwareProfileMemory`, `DeltacloudHardwareProfileStorage`, `DeltacloudKeyname`, and `DeltacloudUserData`. These attributes are set by Condor when the instance runs: `DeltacloudAvailableActions`, `DeltacloudPrivateNetworkAddresses`, `DeltacloudPublicNetworkAddresses`. See section 5.3.9 for details of running jobs under Deltacloud, and see section 10 for definitions of these job ClassAd attributes.
- The configuration variable `JAVA_MAXHEAP_ARGUMENT` has been removed. This means that Java universe jobs will now run with the JVM's default maximum heap setting, unless specified otherwise by the administrator using the configuration of `JAVA_EXTRA_ARGUMENTS` , or by the job via **java_vm_args** in the submit description file as described in section 2.8.
- The configuration variable `TRUST_UID_DOMAIN` was set to `True` in the default `condor_config.local` in the rpm and debian packages. This is no longer the case. This setting will therefore use the default value `False`.
- The configuration variable `NEGOTIATOR_INTERVAL` was set to 20 in the default `condor_config.local` in the rpm and debian packages. This is no longer the case. This setting therefore will use the default value 60.

Bugs Fixed:

- Fixed a bug in *condor_dagman* that caused it to fail when in recovery mode in the face of a specific combination of node job failures with retries.
- Fixed a bug that resulted in the spooled user log not being fetched by *condor_transfer_data*. Prior to Condor version 7.5.4, this problem affected spooled jobs submitted with an explicit list of output files to transfer. In Condor version 7.5.4, this problem also affected spooled jobs that auto-detected output files.
- When a job is submitted with the **-spool** option to *condor_submit*, the *condor_schedd* now correctly writes the submit event to the user log in its spool directory. Previously, it would try to write the event using the user log path given to *condor_submit* by the user, which *condor_submit* may not have access to.

- Fixed a file descriptor leak in the *condor_vm-gahp*. The leak would cause the daemon to fail if a VMware job ran for more than 16 hours on a Linux machine.
- Fixed a bug in *condor_dagman* that caused it to treat any dollar sign in the log file name of a node job's submit description file as an illegal *condor_dagman* macro. Now only the sequence of characters `$(` delimits a macro.

Known Bugs:

- There are two known issues related to the automatic creation of checkpoints with the Condor checkpointing library in Condor version 7.6.0. The first is that compression of standalone checkpoints is disabled for 32-bit binaries. It was always disabled previously, for 64-bit binaries. A standalone checkpoint is one that is run outside of Condor's standard universe. The second problem has to do with compressed 32-bit checkpoint files within the standard universe. If a user requests a compressed 32-bit checkpoint in the standard universe, the resulting checkpoint will not be compressed. As with standalone checkpoints, this has never been supported in 64-bit binaries. We hope to fix both problems in Condor version 7.6.1.

Additions and Changes to the Manual:

- None.

8.4 Development Release Series 7.5

This is the development release series of Condor. The details of each version are described below.

Version 7.5.6

Release Notes:

- Condor version 7.5.6 released on March 21, 2011.
- What used to be known as the *condor_startd* and *condor_schedd* cron mechanisms are now collectively called *Daemon ClassAd Hooks*. The significant changes in this Condor version 7.5.6 release are given in the New Features section.
- In the release directory, the subdirectory `lib/glite/` has been moved to `libexec/glite/`.
- This development series of Condor is no longer officially released for the platforms PowerPC AIX, PowerPC-64 SLES 9, PowerPC MacOS 10.4, Solaris 5.9 on all architectures, Solaris 5.10 on all architectures, Itanium IA64 RHEL 3, PS3 (PowerPC) YDL 5.0, and x86 Debian 4.

- Support for GCB has been removed.
- The default Unix Sys-V init script has been completely reworked. In addition to new features, this changes the following:
  - The default location of the Condor configuration file is now `/etc/condor/condor_config`. This location can be changed by editing the `sysconfig` file or the init script itself.
  - The default location of the Condor installation is now `/usr/`, with binaries in `/usr/bin` and `/usr/sbin`. These locations can also be changed by editing the `sysconfig` file or the init script itself.

New Features:

- Condor no longer relies on DNS to determine its IP address. Instead, it examines the list of system network devices.
- *condor_dagman* now gives a warning if a node category has no nodes assigned to it or no throttle set.
- *condor_dagman* now has a `$MAX_RETRIES` macro for PRE and POST script arguments. Also, *condor_dagman* now prints a warning if an unrecognized macro is used for a PRE or POST script argument. See 66 for details.
- The *condor_schedd* is now more efficient in handling the exit of *condor_shadow* processes, when there are large numbers of *condor_shadow* processes.
- Condor's Chirp protocol has been updated with new commands. The Chirp C++ client and *condor_chirp* command are updated to use the new commands. See section 9 for details on the new commands.
- The Daemon ClassAd Hooks mechanism is described in section 4.4.3, with configuration variables defined in section 3.3.37. The mechanism has the following new features:
  - The *condor_startd*'s benchmarks are no longer hard coded into the *condor_startd*. Instead, the benchmarks are now implemented via the Daemon ClassAd Hooks mechanism. Two new programs are shipped with Condor version 7.5.6: *condor_mips* and *condor_kflops*. These programs are in the `libexec` directory). They implement the original mips and kflops benchmarks for this new implementation. Additional benchmarks can now easily be implemented; the list of benchmarks is controlled via the new `BENCHMARKS_JOBLIST` configuration variable.
  - Several fixes to the the mips and kflops benchmarks should increase the reproducibility of their results.
  - Two new job types have been implemented in the Daemon ClassAd Hooks mechanism. They are called `OneShot` and `OnDemand`. Currently, `OnDemand` is used only by the new `BENCHMARKS` mechanism.

- *condor_dagman* now prints out all boolean configuration variable values as `True` or `False`, instead of 1 or 0 within the `dagman.out` file.
- Because of the new `DAGMAN_VERBOSITY` configuration setting (see section 3.3.26), the **-debug** flag is no longer propagated from a top-level DAG to a sub-DAG; furthermore, **-debug** is no longer set in a `.condor.sub` file unless it is set on the *condor_submit_dag* command line.
- When job ClassAd attributes are modified via *condor_qedit*, the changes are now propagated to the *condor_shadow* and *condor_gridmanager*. This allows a user's changes to the job ClassAd to affect the job policy expressions while the job is managed by these daemons.
- Several improvements for CREAM grid jobs:
  - CREAM commands are retried if the server closes the connection prematurely.
  - All jobs going to a CREAM server share the same lease handle.
  - Multiple CREAM status requests for single jobs are now batched into a single command to the server.
  - When there are too many commands to be issued to a CREAM server simultaneously, new job submissions have lower priority than commands operating on existing jobs.
- The new script *condor_gather_info*, located in `bin/`, creates reports with information from a Condor pool about a specific job ID. It also gathers some understanding of the pool under which it runs.
- Added support for hierarchical accounting groups and group quotas.
- *condor_q* -better-analyze now identifies jobs that have not yet been considered by matchmaking, instead of characterizing them as not matching *for unknown reasons*.
- The default Unix Sys-V init script has been completely reworked. The new version should now work on all Unix and Linux systems. Major features and changes in the new script:
  - Supports the use of a Linux-style `sysconfig` file
  - Supports the use of a Linux-style PID file
  - Supports the following commands:
    - start
    - stop
    - restart
    - try-restart
    - reload
    - force-reload
    - status
  - The default location of the Condor configuration file is now `/etc/condor/condor_config`. This location can be changed by editing the `sysconfig` file or the init script itself.

  - The default location of the Condor installation is now `/usr/`, with binaries in `/usr/bin` and `/usr/sbin`. These locations can be changed by editing the `sysconfig` file or the init script itself.

Configuration Variable and ClassAd Attribute Additions and Changes:

- The default value of configuration variable `GLITE_LOCATION` has changed to `$(LIBEXEC)/glite`. This reflects the change made in the layout of the Condor release files.
- Values for configuration variables `NETWORK_INTERFACE` and `PRIVATE_NETWORK_INTERFACE` may now specify a network device name or an IP address. The asterisk character (`*`) may be used as a wild card in either a name or IP address. This makes it easier to apply the same configuration to a large number of machines, because the IP address does not have to be customized for each host.
- The new configuration variable `DELEGATE_JOB_GSI_CREDENTIALS_LIFETIME` specifies the maximum number of seconds for which delegated job proxies should be valid. The default is one day. A value of 0 indicates that the delegated proxy should be valid for as long as allowed by the credential used to create the proxy; this was the behavior in previous releases of Condor. This configuration variable currently only applies to proxies delegated for non-grid jobs and Condor-C jobs. It does not currently apply to globus grid jobs. The job may override this configuration variable by using the **delegate_job_GSI_credentials_lifetime** submit description file command.
- The new configuration variable `DELEGATE_JOB_GSI_CREDENTIALS_REFRESH` specifies a floating point number between 0 and 1 that indicates when delegated credentials with limited lifetime should be renewed, as a fraction of the delegated lifetime. The default is 0.25.
- The new configuration variable `SHADOW_CHECKPROXY_INTERVAL` specifies the number of seconds between tests to see if the job proxy has been updated or should be refreshed. The default is 600 (10 minutes). Previously, the *condor_shadow* checked for proxy updates once per minute.
- Daemon ClassAd Hooks no longer support what was identified as the *old* syntax. Due to this, variables `STARTD_CRON_JOBS` and `HAWKEYE_JOBS` no longer exist. In previous versions of Condor, the *condor_startd* would issue a warning if this syntax was found, but, starting with 7.5.6, any use of these macros will be ignored.
- New configuration variables `DAGMAN_VERBOSITY` , `DAGMAN_MAX_PRE_SCRIPTS` , `DAGMAN_MAX_POST_SCRIPTS` , and `DAGMAN_ALLOW_LOG_ERROR` are defined in section 3.3.26.
- The new configuration variable `STARTD_PUBLISH_WINREG` can contain a list of Windows registry key names, whose values will be published in the *condor_startd* daemon's ClassAd.
- The new configuration variable `CONDOR_VIEW_CLASSAD_TYPES` is a string list that specifies the types of the ClassAds that will be forwarded to the location defined by `CONDOR_VIEW_HOST`. See the definition at section 3.3.16.

- Added a **-local-name** command line option to *condor_config_val* to inspect the values of attributes that use local names.

Bugs Fixed:

- Fixed a bug for parallel universe jobs, introduced in Condor version 7.5.5, where the *condor_schedd* would crash under certain conditions when a parallel job was removed or exited.
- Fixed a memory leak in the *condor_quill* daemon.
- Fixed a problem in Condor version 7.5.5 release, in which binaries used for the grid universe's pbs and lsf grid types were not marked as executable.
- Fixed a bug introduced in Condor version 7.5.5 that caused running **vanilla**, **java**, and **vm** universe jobs to leave the queue when held.
- A bug has been fixed that caused SOAP transactions in the *condor_schedd* daemon to result in a log message of the form

  ```
  Timer <X> not found
  ```

  This bug is not known to have produced any other undesired behaviors.
- The job ClassAd attribute `JobLeaseDuration` is now set appropriately when a Condor-C job is forwarded to a remote pool. Previously, a default value was not supplied, causing jobs to be unnecessarily killed if the submit and execute machines temporarily lost contact with each other.
- Fixed a bug that caused *condor_dagman* to sometimes falsely report that a cycle existed in a DAG.
- Using the *condor_hold* command on a Windows platform job managed by *condor_dagman* no longer removes the node job of the DAG. This behavior on Windows now matches the behavior on other platforms.
- Using the *condor_hold* command followed by the *condor_remove* command on a job managed by *condor_dagman* now removes node jobs of the DAG, rather than leaving them as orphans.
- A bug has been fixed in the *condor_config_val* program, which caused it to try to contact the *condor_collector* before printing usage information, if the command line was syntactically incorrect.
- A bug has been fixed that caused Condor daemons to crash in response to DNS look up failures when receiving connections. The crash occurred during authorization of the new connection. This problem was introduced in Condor version 7.5.4.
- Fixed a bug that caused *condor_submit* to silently ignore parts of attribute values if an equals sign was omitted.

- Starting in Condor version 7.5.5, the *condor_schedd* daemon would sometimes generate an error message and exit abnormally when shutting down. The error message contained the following text:

  ```
  ERROR ``Assertion ERROR on (m_ref_count == 0)''
  ```

- Changes to the *condor_negotiator* daemon's address were not published to the *condor_collector* until the *condor_negotiator* daemon was reconfigured or restarted. This was a problem in some situations when using *condor_shared_port*.
- A bug introduced in 7.5.5 resulted in failure to advance the flocking level due to lack of activity from one of the negotiators in the flocking list.
- Fixed a Windows-specific problem where the main daemon loop can get into a state where it is busy waiting.
- Fixed *condor_schedd* exception on shutdown caused by bad reference count.
- Releases of Condor with versions from 7.5.0 to 7.5.5 incorrectly implemented the macro language used for configuration with variables having `LOCAL.` at the prefix. This was a regression from the Condor 7.4 series. It is now fixed and the functionality has been restored.

Known Bugs:

- If a cycle exists in the set of jobs to be removed defined by the job ClassAd attribute `OtherJobRemoveRequirements`, removing any of the jobs in the set will cause the *condor_schedd* to go into an infinite loop. `OtherJobRemoveRequirements` is defined on page 909.
- In a *condor_dagman* workflow, if a splice contains nothing but another splice, parsing the DAG will fail. This can be worked around by putting any non-splice job, including a DAG-level NOOP job, into the offending splice. This bug has apparently existed since the splice feature was introduced in *condor_dagman*.
- If an individual Daemon ClassAd Hook manager is not *named*, the jobs under it will attempt to use incorrectly named configuration variables. For example, the following correct configuration will *not* work, because the Daemon ClassAd Hook manager will fail to look up the job's executable variable, given the error in configuration variable naming:

  ```
  STARTD_CRON_JOBLIST = TEST
  ...
  STARTD_CRON_TEST_MODE       = periodic
  STARTD_CRON_TEST_EXECUTABLE = $(LIBEXEC)/test
  ...
  ```

  Condor version 7.5.6 and all previous 7.x Condor versions will incorrectly name the variables from this example `STARTD_TEST_MODE` and `STARTD_TEST_EXECUTABLE` instead. If instead, the Daemon ClassAd Hook Manager had been named, using the no-longer-supported `STARTD_CRON_NAME`, the code works as expected. For example:

```
STARTD_CRON_NAME = HAWKEYE
HAWKEYE_JOBLIST  = TEST
...
HAWKEYE_TEST_MODE       = periodic
HAWKEYE_TEST_EXECUTABLE = $(LIBEXEC)/test
...
```

This *old* behavior is, as of Condor version 7.5.6, documented as unsupported and is going away, primarily because it is confusing. But, for this release, it still works. It is believed that this same behavior exists in all 7.x releases of Condor, but because the naming feature is used, the incorrect behavior went undetected.

This affects the `STARTD_CRON` and `SCHEDD_CRON` Daemon ClassAd Hook managers, and will be fixed in Condor version 7.6.0.

Additions and Changes to the Manual:

- None.

Version 7.5.5

Release Notes:

- Condor version 7.5.5 released on January 26, 2011.
- This version of Condor uses a different layout in the spool directory for storing files belonging to jobs that are in the queue. Conversion of the spool directory is automatic when upgrading, but be aware that *downgrading to a previous version of Condor requires extra effort*. The procedure for downgrading is either to drain all jobs with spooled files from the queue, or to manually convert the spool back to the older format. To manually convert back to the older format, stop Condor and back up the spool directory in case of problems. Then move all subdirectories matching the form `$(SPOOL)/<#>/<#>/cluster<#>.proc<#>.subproc<#>` into `$(SPOOL)`. Also do this for any files of the form `$(SPOOL)/<#>/cluster<#>.ickpt.subproc<#>`. Edit `$(SPOOL)/job_queue.log` with a text editor, and change all references to the old paths to the new paths. Then, remove `$(SPOOL)/spool_version`. Finally, start up Condor.
- For those who compile Condor from the source code rather than using packages of pre-built executables, be aware that in this release Condor is built using *cmake* instead of *imake*. See the `README.building` file for the new instructions on how to build Condor.
- This release note serves to remind users that as of Condor version 7.5.1, the RPMs come with native packaging. Therefore, items are in different locations, as given by FHS locations, such as `/usr/bin`, `/usr/sbin`, `/etc`, and `/var/log`. Please see section 3.2.6 for installation documentation.

- Quill is now available only within the source code distribution of Condor. It is no longer included in the builds of Condor provided by UW, but it is available as a feature that can be enabled by those who compile Condor from the source code. Find the code within the `condor_contrib` directory, in the directories `condor_tt` and `condor_dbmsd`.
- The AIX 5.2 packages in this release have been found to be incompatible with AIX 5.3.
- We are planning to drop support for AIX. Please contact us if this is a problem for you.
- The directory structure within the Unix tar file package of Condor has changed. Previously, the tar file contained a top level directory named `condor-<version>`. The top level directory is now the same as the tar file name, but without the `.tar.gz` extension.
- On Unix platforms, the following executables used to be located in both the `sbin` and `bin` directories, but are now only located in the `bin` directory: *condor*, *condor_checkpoint*, *condor_reschedule*, and *condor_vacate*.
- The size of the Condor installation has increased by as much as 60% compared to Condor version 7.5.4. We hope to eliminate most of this increase in Condor version 7.5.6.
- Previously, packages containing debug symbols were available for many Unix platforms. In this release, the debug packages contain full, 'unstripped' executables instead of just the debug symbols.
- The contents of the Windows zip and MSI packages of Condor have changed. The `lib` and `libexec` folders no longer exist, and all contents previously within them are now in `bin`. *condor_setup* and *condor_set_acls* have been moved from the top level directory into `bin`.
- The Windows MSI installer for Condor version 7.5.5 requires that all previous MSI installations of Condor be uninstalled. Before uninstalling previous versions, make backup copies of configuration files. Any settings that need to be preserved must be reapplied to the configuration of the new installation.
- The following list itemizes changes included in this Condor version 7.5.5 release that belong to Condor version 7.4.5. That stable series version will not yet have been released as this development version is released.
  - *condor_dagman* now prints a message in the `dagman.out` file whenever it truncates a node job user log file. *condor_dagman* now prints additional diagnostic information in the case of certain log file errors.
  - Fixed a bug in which a network disconnect between the submit machine and execute machine during the transfer of output files caused the *condor_starter* daemon to immediately give up, rather than waiting for the *condor_shadow* to reconnect. This problem was introduced in Condor version 7.4.4.
  - Fixed a bug in which if *condor_ssh_to_job* attempted to connect to a job while the job's input files were being transferred, this caused the file transfer to fail, which resulted in the job returning to the idle state in the queue.
  - In privsep mode, the transfer of output failed if a job's execute directory contained symbolic links to non-existent paths.

New Features:

- Negotiation is now handled asynchronously in the *condor_schedd* daemon. This means that the *condor_schedd* remains responsive during negotiation and is less prone to falling behind on communication with *condor_shadow* processes.
- Improved monitoring and avoidance of a *lock convoy* problem observed when there were more than 30,000 *condor_shadow* processes. At this scale, locking the *condor_shadow* daemon's log on each write to the log file has been observed on Linux platforms to sometimes result in a situation where the system does very little productive work, and is instead consumed by rapid context switching between the *condor_shadow* daemons that are waiting for the lock.
- On Linux platforms, if the *condor_schedd* daemon's spool directory is on an ext3 file system, Condor can now scale to a larger number of spooled jobs. Previously, Condor created two subdirectories within the spool directory for each spooled job and for each running job. The ext3 file system only supports 31,997 subdirectories. This effectively limited the number of spooled jobs to less than 16,000. Now, Condor creates a hierarchy of subdirectories within the spool directory, to increase the limit on the number of spooled jobs in ext3 to 320,000,000, which is likely to be larger than other limits on the size of the job queue, such as memory.
- The *condor_shadow* daemon uses less memory than it has since Condor version 7.5.0. Memory usage should now be similar to the 7.4 series.
- The *condor_dagman* and *condor_submit_dag* command-line flag **-DumpRescue** causes the dump of an incomplete Rescue DAG, when the parsing of the DAG input file fails. This may help in figuring out what went wrong. See section 2.10.7 for complete details on Rescue DAGs.
- *condor_dagman* now has the capability to create the `jobstate.log` file needed for the Pegasus workflow manager. See section 2.10.11 for details.
- *condor_wait* can now work on jobs with logs that are only readable by the user running *condor_wait*. Previously, write access to the job's user log was required.
- Added a new value for the job ClassAd attribute `JobStatus`. The `TRANSFERRING_OUTPUT` status is used when transferring a job's output files after the job has finished running. Jobs with this status will have their `JobStatus` attribute set to 6. The standard *condor_q* display will show the job's status as >.

Configuration Variable and ClassAd Attribute Additions and Changes:

- The new configuration variable `LOCK_DEBUG_LOG_TO_APPEND` controls whether a daemon's debug lock is used when appending to the log. When the default value of `False`, the debug lock is only used when rotating the log file. When `True`, the debug lock is used when writing to the log as well as when rotating the log file. See section 3.3.4 for the complete definition.

- The new configuration variable `LOCAL_CONFIG_DIR_EXCLUDE_REGEXP` may be set to a regular expression that specifies file names to ignore when looking for configuration files within the directories specified via `LOCAL_CONFIG_DIR`. See section 3.3.3 for the complete definition.

Bugs Fixed:

- In previous versions of Condor, the *condor_starter* could not write the `.machine.ad` and `.job.ad` files to the `execute` directory when PrivSep was enabled. This has now been fixed, and these files are correctly emitted in all cases.
- Since Condor version 7.5.2, the speed of *condor_q* was not as high as earlier 7.5 and 7.4 releases, especially when retrieving large numbers of jobs. Viewing 100K jobs took about four times longer. This release fixes the performance, making it about the same as before Condor version 7.5.2.
- A bug introduced in Condor version 7.5.4 prevented parallel universe jobs with multiple **queue** statements in the submit description file from working with *condor_dagman*. This is now fixed.
- Improved the way Condor daemons send heartbeat messages to their parent process. This resolves a problem observed on busy submit machines using the *condor_shared_port* daemon. The *condor_master* daemon sometimes incorrectly determined that the *condor_schedd* was hung, and would kill and restart it.
- When the configuration variable `NOT_RESPONDING_WANT_CORE` is `True`, the *condor_master* daemon now follows up with a `SIGKILL`, if the child process does not exit within ten minutes of receiving `SIGABRT`. This addresses observed cases in which the child process hangs while writing a core file.
- Host name-based authorization failed in Condor version 7.5.4 under Mac OS X 10.4, because look ups of the host name for incoming connections failed.
- A bug introduced in Condor version 7.5.0 caused the attributes `MyType` and `TargetType` in offline ClassAds to get set to `"(unknown type)"` when the offline ClassAd was matched to a job.
- *condor_dagman* now excepts in the case of certain log file errors, when continuing would be likely to put *condor_dagman* into an incorrect internal state.
- Fixed a bug that caused DAG node jobs to have their coredumpsize limit set according to the `CREATE_CORE_FILES` configuration variable, rather than the user's coredumpsize limit.
- Fixed a case introduced in Condor version 7.5.4 on Windows platforms, in which the following spurious log message was produced:

```
SharedPortEndpoint: Destructor: Problem in thread shutdown notification: 0
```

- Since Condor version 7.4.1, Condor-C jobs submitted without file transfer enabled could fail with the following error in the *condor_starter* log:

  ```
  FileTransfer: DownloadFiles called on server side
  ```

- Fixed a memory leak caused by use of the ClassAd `eval()` function. This problem was introduced in Condor version 7.5.2.
- Fixed a bug that could cause the *condor_negotiator* daemon to crash when groups are configured with `GROUP_QUOTA_DYNAMIC_<group_name>`, or when `GROUP_QUOTA_<group_name>` is not defined to be something greater than 0.
- Fixed a bug that caused random characters to appear for the value of `AuthMethods` when logging with `D_FULLDEBUG` and `D_SECURITY` enabled. This problem was introduced in Condor version 7.5.3.
- Fixed a memory leak in the *condor_schedd* introduced in Condor version 7.5.4.
- Fixed a problem introduced in Condor version 7.5.4 that could cause the *condor_schedd* daemon to enter an infinite loop while in the process of shutting down. For the problem to happen, it was necessary for flocking to have been enabled.
- Configuration variable `SCHEDD_QUERY_WORKERS` was effectively ignored when *condor_q* authenticated itself to the *condor_schedd*. The query was always processed in the main *condor_schedd* process rather than in a sub-process. This problem has existed since before Condor version 7.0.0.
- Fixed a problem affecting jobs that store their output in the *condor_schedd*'s spool directory. The problem affected jobs that include an empty directory in their list of output files to transfer. This problem was introduced in Condor version 7.5.4, when support for the transfer of directories was added.
- Fixed a problem affecting the *condor_master* daemon since Condor version 7.5.3. The *condor_master* daemon would crash if it was instructed to shut down a daemon that was not currently running, but which was waiting to be restarted.
- Fixed a bug in *condor_submit* that prevented the submission of multiple **vm** universe jobs in a single submit file.
- Fixed a bug in the *condor_schedd* that could cause it to temporarily under count the number of running local and scheduler universe jobs. In Condor version 7.5.4, this under counting could cause the daemon to crash.
- Fixed a bug that could cause the *condor_gridmanager* to crash if a GAHP server did not behave as expected on start up.
- Improved the hold reason reported in several failure cases for CREAM grid jobs.
- The `KFlops` attribute reported by

  ```
  condor_status -run -total
  ```

could erroneously be reported as negative. This has been fixed.

- Since Condor version 7.5.4, the refreshing of the proxy for the job in the remote queue did not work in Condor-C. Therefore, if the original job proxy expired, the job was halted and put on hold, even if the proxy had been renewed on the submit machine.

Known Bugs:

- In Condor version 7.5.5, when a running job is put on hold, the job is removed from the job queue.

Additions and Changes to the Manual:

- None.

Version 7.5.4

Release Notes:

- Condor version 7.5.4 released on October 20, 2010.
- All of the bug fixes and features which are in Condor version 7.4.4 are in this 7.5.4 release.
- The release now contains all header files necessary to compile code that uses the job log reading and writing utilities contained in libcondorapi. Some headers were missing starting in Condor 7.5.1.

New Features:

- Concurrency limits now work with parallel universe jobs scheduled by the dedicated scheduler.
- Transfer of directories is now supported by **transfer_input_files** and **transfer_output_files** for non-grid universes and Condor-C. The auto-selection of output files, however, remains the same: new directories in the job's output sandbox are *not* automatically selected as outputs to be transferred.
- Paths other than simple file names with no directory information in **transfer_output_files** previously did not have well defined behavior. Now, paths are supported for non-grid universes and Condor-C. When a path to an output file or directory is specified, this specifies the path to the file on the execute side. On the submit side, the file is placed in the job's initial working directory and it is named using the base name of the original path. For example, `path/to/output_file` becomes `output_file` in the job's initial working directory. The name and path of the file that is written on the submit side may be modified by using **transfer_output_remaps**.

- The *condor_shared_port* daemon is now supported on Windows platforms.
- Jobs can now by submitted to multiple EC2 servers via the amazon grid type. The server's URL must be specified via the **grid_resource** submit description file command for each job. See section 5.3.7 for details.
- The grid universe's amazon grid type can now be used to submit virtual machine jobs to Eucalyptus systems via the EC2 interface.
- *condor_q* now uses the queue-management API's projection feature when used with **-run**, **-hold**, **-goodput**, **-cputime**, **-currentrun**, and **-io** options when called with no display options or with **-format**.
- Decreased the CPU utilization of *condor_dagman* when it is submitting ready jobs into Condor.
- *condor_dagman* now logs the number of queued jobs in the DAG that are on hold, as part of the DAG status message in the `dagman.out` file.
- *condor_dagman* now logs a note in the `dagman.out` file when the *condor_submit_dag* and *condor_dagman* versions differ, even if the difference is permissible.
- Added the capability for *condor_dagman* to create and periodically rewrite a file that lists the status of all nodes within a DAG. Alternatively, the file may be continually updated as the DAG runs. See section 2.10.10 for details.
- The *condor_schedd* daemon now uses a better algorithm for determining which flocking level is being negotiated. No special configuration is required for the new algorithm to work. In the past, the algorithm depended on DNS and the configuration variables `NEGOTIATOR_HOST` and `FLOCK_NEGOTIATOR_HOSTS`. In some networking environments, such as that of a multi-homed central manager, it was difficult to configure things correctly. When wrongly configured, negotiation would be aborted with the message, `Unknown negotiator`. The new algorithm is only used when the *condor_negotiator* is version 7.5.4 or newer. Of course, the *condor_schedd* daemon must still be configured to authorize the *condor_negotiator* daemon at the `NEGOTIATOR` authorization level.
- *condor_advertise* has a new option, **-multiple**, which allows multiple ClassAds to be published. This is more efficient than publishing each ClassAd in a separate invocation of *condor_advertise*.
- The *condor_job_router* is no longer restricted to routing only vanilla universe jobs. It also now automatically avoids recursively routing jobs.
- The *condor_schedd* now writes the submit event to the user job log. Previously, *condor_submit* wrote the event.
- The *condor_schedd* daemon now scales better when there are many job auto clusters.
- The *condor_q* command with option **-run**, **-hold**, **-goodput**, **-cputime**, **-currentrun** or **-io** is now much more efficient in its communication with the *condor_schedd*.

Configuration Variable and ClassAd Attribute Additions and Changes:

- The new configuration variable SOAP_SSL_SKIP_HOST_CHECK can be used to disable the standard check that a SOAP server's host name matches the host name in its X.509 certificate. This is useful when submitting grid type amazon jobs to Eucalyptus servers, which often have certificates with a host name of localhost.
- Added default values for <SUBSYS>_LOG configuration variables. If a <SUBSYS>_LOG configuration variable is not set in files condor_config or condor_config.local, it will default to $(LOG)/<SUBSYS>LOG.
- The new job ClassAd attribute CommittedSuspensionTime is a running total of the number of seconds the job has spent in suspension during time in which the job was not evicted without a checkpoint. This complements the existing attribute CumulativeSuspensionTime, which includes all time spent in suspension, regardless of job eviction.
- The new job ClassAd attributes CommittedSlotTime and CumulativeSlotTime are just like CommittedTime and RemoteWallClockTime respectively, except the new attributes are weighted by the SlotWeight of the machine(s) that ran the job.
- The new configuration variable SYSTEM_JOB_MACHINE_ATTRS specifies a list of machine attributes that should be recorded in the job ClassAd. The default attributes are Cpus and SlotWeight. When there are multiple run attempts, history of machine attributes from previous run attempts may be kept. The number of run attempts to store is specified by the new configuration variable SYSTEM_JOB_MACHINE_ATTRS_HISTORY_LENGTH , which defaults to 1. A machine attribute named X will be inserted into the job ClassAd as an attribute named MachineAttrX0. The previous value of this attribute will be named MachineAttrX1, the previous to that will be named MachineAttrX2, and so on, up to the specified history length. Additional attributes to record may be specified on a per-job basis by using the new **job_machine_attrs** submit file command. The history length may also be extended on a per-job basis by using the new submit file command **job_machine_attrs_history_length**.
- The new configuration variable NEGOTIATION_CYCLE_STATS_LENGTH specifies how many recent negotiation cycles should be included in the history that is published in the *condor_negotiator*'s ClassAd. The default is 3. See page 228 for the definition of this configuration variable, and see page 923 for a list of attributes that are published.
- The configuration variable FLOCK_NEGOTIATOR_HOSTS is now optional. Previously, the *condor_schedd* daemon refused to flock without this setting. When this is not set, the addresses of the flocked *condor_negotiator* daemons are found by querying the flocked *condor_collector* daemons. Of course, the *condor_schedd* daemon must still be configured to authorize the *condor_negotiator* daemon at the NEGOTIATOR authorization level. Therefore, when using host-based security, FLOCK_NEGOTIATOR_HOSTS may still be useful as a macro for inserting the negotiator hosts into the relevant authorization lists.

- The configuration variable FLOCK_HOSTS is no longer used. For backward compatibility, this setting used to be treated as a default for FLOCK_COLLECTOR_HOSTS and FLOCK_NEGOTIATOR_HOSTS.
- The configuration variable AMAZON_EC2_URL is now only used for previously-submitted jobs when upgrading Condor to version 7.5.4 or beyond. New grid type amazon jobs must specify which EC2 service to use by setting the **grid_resource** submit description file command.
- The new job ClassAd attribute NumPids is the total number of child processes a running job has.
- The new configuration variable DAGMAN_MAX_JOB_HOLDS specifies the maximum number of times a DAG node job is allowed to go on hold. See section 3.3.26 for details.
- The configuration variable STARTD_SENDS_ALIVES now only needs to be set for the *condor_schedd* daemon. Also, the default value has changed to True.
- The job ClassAd attributes **amazon_user_data** and **amazon_user_data_file** can now both be used for the same job. When both are provided, the two blocks of data are concatenated, with the value of the one specified by **amazon_user_data** occurring first.
- The new configuration variable GRAM_VERSION_DETECTION can be used to disable Condor's attempts to distinguish between gt2 (GRAM2) and gt5 (GRAM5) servers. The default value is True. If set to False, Condor trusts the gt2 or gt5 value provided in the job's **grid_resource** attribute.
- The new job ClassAd attribute ResidentSetSize is an integer measuring the amount of physical memory in use by the job on the execute machine in kilobytes.
- The new job ClassAd attribute X509UserProxyExpiration is an integer representing when the job's X.509 proxy credential will expire, measured in the number of seconds since the epoch (00:00:00 UTC, Jan 1, 1970).
- The new configuration variable SCHEDD_CLUSTER_MAXIMUM_VALUE is an upper bound on assigned job cluster ids. If set to value M, the maximum job cluster id assigned to any job will be $M - 1$. When the maximum id is reached, assignment of cluster ids will wrap around back to SCHEDD_CLUSTER_INITIAL_VALUE. The default value is zero, which does not set a maximum cluster id.
- The default value of configuration variable MAX_ACCEPTS_PER_CYCLE has been changed from 1 to 4.
- The configuration variable NEW_LOCKING , introduced in Condor version 7.5.2, has been changed to CREATE_LOCKS_ON_LOCAL_DISK and now defaults to True.

Bugs Fixed:

- Fixed a bug that occurred with x64 flavors of the Windows operating system. Condor was setting the default value of `Arch` to `INTEL` when it should have been `X86_64`. This was a consequence of the fact that the Condor runs in the WOW64 sandbox on 64-bit Windows. This was fixed so that `Arch` would contain the value for the native architecture rather than the WOW64 sandbox architecture.
- Fixed a bug in the user privilege switching code in Windows that caused the *condor_shadow* daemon to except when the *condor_schedd* daemon attempted to re-use it.
- Fixed the output in the *condor_master* daemon log file to be clearer when an authorized user tries to use *condor_config_val* **-set** and `ENABLE_PERSISTENT_CONFIG` is `False`. The previous output implied that the operation succeeded when, in fact, it did not.
- Since Condor version 7.5.2, the following *condor_job_router* features were effectively non-functional: `UseSharedX509UserProxy`, `JobShouldBeSandboxed`, and `JobFailureTest`.
- The submit description file command **copy_to_spool** did not work properly in Condor version 7.5.3. When sending the executable to the execute machine, it was transferred from the original path rather than from the spooled copy of the file.
- When output files were auto-selected and spooled, Condor-C and *condor_transfer_data* would copy back both the output files and all other contents of the job's spool directory, which typically included the spooled input and the user log. Now, only the output files are retrieved. To adjust which files are retrieved, the job attribute `SpooledOutputFiles` can be manipulated, but this typically should be managed by Condor.
- The *condor_master* daemon now invalidates its ClassAd, as represented in the *condor_collector* daemon, before it shuts down.
- Fixed a bug that caused **vm** universe jobs to not run if the VMware `.vmx` file contained a space.
- Fixed a bug introduced in Condor version 7.5.1 that caused integers in ClassAd expressions that had leading zeros to be read as octal (base eight).
- Fixed a bug introduced in Condor version 7.5.1 that did not recognize a semicolon as a separator of function arguments in ClassAds.
- Fixed a bug that caused integers larger than $\pm 2^{31}$ in a ClassAd expression to be parsed incorrectly. Now, when these integers are encountered, the largest 32-bit integer (with matching sign) is used.
- Fixed a bug that caused the *condor_gridmanager* to exit when receiving badly-formatted error messages from the *nordugrid_gahp*.
- Fixed a problem affecting the use of version 7.5.3 *condor_startd* and *condor_master* daemons in a pool with a *condor_collector* from before version 7.5.2. On shutdown, the *condor_startd* and the *condor_master* caused all *condor_startd* and *condor_master* ClassAds, respectively, to be removed from the *condor_collector*.

- Fixed a bug that caused delegation of an X.509 RFC proxy between two Condor processes to fail.
- Fixed a bug in *condor_submit* that would cause failures if a file name containing a space was used with the submit description file commands **append_files**, **jar_files** or **vmware_dir**.
- Fixed a bug that could cause the *condor_gridmanager* to lock up if a GAHP server it was using wrote a large amount of data to its `stderr`.
- Fixed a bug that could cause the *condor_gridmanager* to wrongly conclude that a `gt2` (that is, GRAM2) server was a `gt5` (that is, GRAM5) server. Such a conclusion can be disastrous, as Condor's mechanisms to prevent overloading a `gt2` server are then disabled. The new configuration variable `GRAM_VERSION_DETECTION` can be used to disable Condor's attempts to distinguish between the two.
- Fixed a bug introduced in Condor version 7.5.3. When file transfer failed for a **grid** universe job of grid type cream, Condor would write a hold event to the job log, but not actually put the job on hold.
- Fixed a bug in the *condor_gridmanager* that could cause it to crash while handling cream grid type jobs destined for different resources.
- Fixed a bug that prevented the *condor_shadow* from managing additional jobs after its first job completed when `SEC_ENABLE_MATCH_PASSWORD_AUTHENTICATION` was set to `True`.
- The timestamps in the log defined by `PROCD_LOG` now print the real time.
- Fixed how some daemons advertise themselves to the *condor_collector*. Now, all daemons set the attribute `MyType` to indicate what type of daemon they are.
- *condor_chirp* no longer crashes on a put operation, if the remote file name is omitted.
- Fixed the packaging of Hadoop File System support in Condor. This includes updating to HDFS 0.20.2 and making the HDFS web interface work properly.
- Condor no longer tries to invoke *glexec* if the job's X.509 proxy is expired.

Known Bugs:

- Using host names for host-based authentication, such as in the definitions of configuration variables `ALLOW_*` and `DENY_*`, does not work on Mac OS X 10.4. Later versions of the OS are not affected. As a work around, IP addresses can be used instead of host names.

Additions and Changes to the Manual:

- None.

Version 7.5.3

Release Notes:

- Condor version 7.5.3 released on June 29, 2010.

New Features:

- *condor_q* **-analyze** now notices the **-l** option, and if both are given, then the analysis prints out the list of machines in each analysis category.
- The behavior of macro expansion in the configuration file has changed. Previously, most macros were effectively treated as undefined unless explicitly assigned a value in the configuration file. Only a small number of special macros had pre-defined values that could be referred to via macro expansion. Examples include `FULL_HOSTNAME` and `DETECTED_MEMORY`. Now, most configuration settings that have default values can be referred to via macro expansion. There are a small number of exceptions where the default value is too complex to represent in the current implementation of the configuration table. Examples include the security authorization settings. All such configuration settings will also be reported as undefined by *condor_config_val* unless they are explicitly set in the configuration file.
- Unauthenticated connections are now identified as `unauthenticated@unmapped`. Previously, unauthenticated connections were not assigned a name, so some authorization policies that needed to distinguish between authenticated and unauthenticated connections were not expressible. Connections that are authenticated but not mapped to a name by the mapfile used to be given the name `auth-method@unmappeduser`, where *auth-method* is the authentication method that was used. Such connections are now given the name `auth-method@unmapped`. Connections that match `*@unmapped` are now forbidden from doing operations that require a user id, regardless of configuration settings. Such operations include job submission, job removal, and any other job management commands that modify jobs.
- There has been a change of behavior when authentication fails. Previously, authentication failure always resulted in the command being rejected, regardless of whether the ALLOW/DENY settings permitted unauthenticated access or not. This is still true if either the client or server specifies that authentication is required. However, if both sides specify that authentication is not required (i.e. preferred or optional), then authentication failure only results in the command being rejected if the ALLOW/DENY settings reject unauthenticated access. This change makes it possible to have some commands accept unauthenticated users from some network addresses while only allowing authenticated users from others.
- Improved log messages when failing to authenticate requests. At least the IP address of the requester is identified in all cases.

- The new submit file command **job_ad_information_attrs** may be used to specify attributes from the job ad that should be saved in the user log whenever a new event is being written. See page 850 for details.
- Administrative commands now support the **-constraint** option, which accepts a ClassAd expression. This applies to *condor_checkpoint*, *condor_off*, *condor_on*, *condor_reconfig*, *condor_reschedule*, *condor_restart*, *condor_set_shutdown*, and *condor_vacate*.
- File transfer plugins can be used for vm universe jobs. Notably, `file://` URLs can be used to allow VM image files to be pre-staged on the execute machine. The submit description file command **vmware_dir** is now optional. If it is not given, then all relevant VMware image files must be listed in **transfer_input_files**, possibly as URLs.
- File transfers for CREAM grid universe jobs are now initiated by the *condor_gridmanager*. This removes the need for a GridFTP server on the client machine.
- Improved the parallelism of file transfers for nordugrid jobs.
- Removed the distinction between regular and full reconfiguration of Condor daemons. Now, all reconfigurations are full and require the WRITE authorization level. *condor_reconfig* accepts but ignores the **-full** command-line option.
- The *batch_gahp*, used for pbs and lsf grid universe jobs, has been updated from version 1.12.2 to 1.16.0.
- *condor_dagman* now prints a message to the `dagman.out` file when it truncates a node job user log file.
- *condor_dagman* now allows node categories to include nodes from different splices. See section 2.10.6 for details.
- *condor_dagman* now allows category throttles in splices to be overridden by higher levels in the DAG splicing structure. See section 2.10.6 for details.
- Daemon logs can now be rotated several times instead of only once into a single `.old` file. In order to do so, the newly introduced configuration variable `MAX_NUM_<SUBSYS>_LOG` needs to be set to a value greater than 1. The file endings will be ISO timestamps, and the oldest rotated file will still have the ending `.old`.

Configuration Variable and ClassAd Attribute Additions and Changes:

- The new configuration variable `JOB_ROUTER_LOCK` specifies a lock file used to ensure that multiple instances of the *condor_job_router* never run with the same values of `JOB_ROUTER_NAME`. Multiple instances running with the same name could lead to mismanagement of routed jobs.
- The new configuration variable `ROOSTER_MAX_UNHIBERNATE` is an integer specifying the maximum number of machines to wake up per cycle. The default value of 0 means no limit.

- The new configuration variable ROOSTER_UNHIBERNATE_RANK is a ClassAd expression specifying which machines should be woken up first in a given cycle. Higher ranked machines are woken first. If the number of machines to be woken up is limited by ROOSTER_MAX_UNHIBERNATE, the rank may be used for determining which machines are woken before reaching the limit.
- The new configuration variable CLASSAD_USER_LIBS is a list of libraries containing additional ClassAd functions to be used during ClassAd evaluation.
- The new configuration variable SHADOW_WORKLIFE specifies the number of seconds after which the *condor_shadow* will exit, when the current job finishes, instead of fetching a new job to manage. Having the *condor_shadow* continue managing jobs helps reduce overhead and can allow the *condor_schedd* to achieve higher job completion rates. The default is 3600, one hour. The value 0 causes *condor_shadow* to exit after running a single job.
- The new configuration variable MAX_NUM_<SUBSYS>_LOG will determine how often the daemon log of SUBSYS will rotate. The default value is 1 which leads to the old behavior of a single rotation into a .old file.

Bugs Fixed:

- Configuration variables with a default value of 0 that were not defined in the configuration file were treated as though they were undefined by *condor_config_val*. Now Condor treats this case like any other: the default value is displayed.
- Starting in Condor version 7.5.1, using literals with a logical operator in a ClassAd expression (for example, 1 || 0) caused the expression to evaluate to the value ERROR. The previous behavior has been restored: zero values are treated as False, and non-zero values are treated as True.
- Starting in Condor version 7.5.0, the *condor_schedd* no longer supported queue management commands when security negotiation was disabled, for example, if SEC_DEFAULT_NEGOTIATION = NEVER.
- Fixed a bug introduced in Condor version 7.5.1. ClassAd string literals containing characters with negative ASCII values were not accepted.
- Fixed a bug introduced in Condor version 7.5.0, which caused Condor to not renew job leases for CREAM grid jobs in most situations.
- Question marks occurring in a ClassAd string are no longer preceded by a backslash when the ClassAd is printed.

Known Bugs:

- None.

Additions and Changes to the Manual:

- None.

Version 7.5.2

Release Notes:

- Condor version 7.5.2 released on April 26, 2010.
- Condor no longer supports SuSE 8 Linux on the Itanium 64 architecture.
- The following submit description file commands are no longer recognized. Their functionality is replaced by the command **grid_resource**.

  grid_type
  globusscheduler
  jobmanager_type
  remote_schedd
  remote_pool
  unicore_u_site
  unicore_v_site

New Features:

- The *condor_schedd* daemon uses less disk bandwidth when logging updates to job ClassAds from running jobs and also when removing jobs from the queue and handling job eviction and *condor_shadow* exceptions. This should improve performance in situations where disk bandwidth is a limiting factor. Some cases of updates to the job user log have also been optimized to be less disk intensive.
- The *condor_schedd* daemon uses less CPU when scheduling some types of job queues. Most likely to benefit from this improvement is a large queue of short-running, non-local, and non-scheduler universe jobs, with at least one idle local or scheduler universe job.
- The *condor_schedd* automatically grants the *condor_startd* authority to renew leases on claims and to evict claims. Previously, this required that the *condor_startd* be trusted for general `DAEMON`-level command access. Now this only requires `READ`-level command access. The specific commands that the *condor_startd* sends to the *condor_schedd* can effectively only operate on the claims associated with that *condor_startd*, so this change does not open up these operations to access by anyone with `READ` access. It reduces the level of trust that the *condor_schedd* must have in the *condor_startd*.

- The *condor_procd*'s log now rotates if logging is activated. The default maximum size is 10Mbytes. To change the default, use the configuration variable `MAX_PROCD_LOG`.
- For Unix systems only, user job log and global job event log lock files can now optionally be created in a directory on a local drive by setting `NEW_LOCKING` to `True`. See section 3.3.4 for the details of this configuration variable.
- *condor_dagman* and *condor_submit_dag* now default to lazy creation of the `.condor.sub` files for nested DAGs. *condor_submit_dag* no longer creates them, and *condor_dagman* itself creates the files as the DAG is run. The previous "eager" behavior can be obtained with a combination of command-line and configuration settings. There are several advantages to the "lazy" submit file creation:
  - The DAG file for a nested DAG does not have to exist until that node is ready to run, so the DAG file can be dynamically created by earlier parts of the top-level DAG (including by the PRE script of the nested DAG node).
  - It is now possible to have nested DAGs within splices, which is not possible with "eager" submit file creation.

Configuration Variable and ClassAd Attribute Additions and Changes:

- The new configuration variable `DAGMAN_GENERATE_SUBDAG_SUBMITS` controls whether *condor_dagman* itself generates the `.condor.sub` files for nested DAGs, rather than relying on *condor_submit_dag* "eagerly" creating them. See section 3.3.26 for more information.
- The new configuration variable `NEW_LOCKING` can specify that job user logs and the global job event log to be written to a local drive, avoiding locking problems with NFS. See section 3.3.4 for the details of this configuration variable.

Bugs Fixed:

- The *condor_job_router* failed to work on SLES 9 PowerPC, AIX 5.2 PowerPC, and YDL 5 PowerPC due to a problem in how it detected EOF in the job queue log.
- When jobs are removed, the *condor_schedd* sometimes did not quickly reschedule a different job to run on the slot to which the removed job had been matched. Instead, it would take up to `SCHEDD_INTERVAL` seconds to do so.
- Fixed a bug introduced in Condor version 7.5.1 that caused the *gahp_server* to crash when first communicating with most gt2 or gt5 GRAM servers.

Known Bugs:

- None.

Additions and Changes to the Manual:

- None.

Version 7.5.1

Release Notes:

- Condor version 7.5.1 released on March 2, 2010.
- Some, but not all of the bug fixes and features which are in Condor version 7.4.2, are in this 7.5.1 release.
- The Condor release is now available as a proper RPM or Debian package.
- Condor now internally uses the version of New ClassAds provided as a stand-alone library (http://www.cs.wisc.edu/condor/classad/). Previously, Condor used an older version of Class-Ads that was heavily tied to the Condor development libraries. This change should be transparent in the current development series. In the next development series (7.7.x), Condor will begin to use features of New ClassAds that were unavailable in Old ClassAds. Section 4.1.1 details the differences.
- HPUX 11.00 is no longer a supported platform.

New Features:

- A port number defined within `CONDOR_VIEW_HOST` may now use a shared port.
- The *condor_master* no longer pauses for 3 seconds after starting the *condor_collector*. However, if the configuration variable `COLLECTOR_ADDRESS_FILE` defines a file, the *condor_master* will wait for that file to be created before starting other daemons.
- In the grid universe, Condor can now automatically distinguish between GRAM2 and GRAM5 servers, that is grid types **gt2** and **gt5**. Users can submit jobs using a grid type of **gt2** or **gt5** for either type of server.
- Grid universe jobs using the CREAM grid system now batch up common requests into larger single requests. This reduces network traffic, increases the number of parallel tasks the Condor can handle at once, and reduces the load on the remote gatekeeper.
- The new submit description file command **cream_attributes** sets additional attribute/value pairs for the CREAM job description that Condor creates when submitting a grid universe job destined for the CREAM grid system.
- The *condor_q* command with option **-analyze** is now performs the same analysis as previously occurred with the **-better-analyze** option. Therefore, the output of *condor_q* with the **-analyze** option has different output than before. The **-better-analyze** option is still recognized and behaves the same as before, though it may be removed from a future version.

- Security sessions that are not used for longer than an hour are now removed from the security session cache to limit memory usage.
- The number of security sessions in the cache is now advertised in the daemon ClassAd as `MonitorSelfSecuritySessions`.
- *condor_dagman* now has the capability to run DAGs containing nodes that are declared to be NOOPs – for these nodes, a job is never actually submitted. See section 2.10.2 for information.
- The submit file attribute **vm_macaddr** can now be used to set the MAC address for vm universe jobs that use VMware. The range of valid MAC addresses is constrained by limits imposed by VMware.
- The *condor_q* command with option **-globus** is now much more efficient in its communication with the *condor_schedd*.

Configuration Variable and ClassAd Attribute Additions and Changes:

- The new configuration variable `STRICT_CLASSAD_EVALUATION` controls whether new or old ClassAd expression evaluation semantics are used. In new ClassAd semantics, an unscoped attribute reference is only looked up in the local ad. The default is False (use old ClassAd semantics).
- The configuration variable `DELEGATE_FULL_JOB_GSI_CREDENTIALS` now applies to all proxy delegations done between Condor daemons and tools. The value is a boolean and defaults to `False`, which means that when doing delegation Condor will now create a limited proxy instead of a full proxy.
- The new configuration variable `SEC_<access-level>_SESSION_LEASE` specifies the maximum number of seconds an unused security session will be kept in a daemon's session cache before being removed to save memory. The default is 3600. If the server and client have different configurations, the smaller one will be used.

Bugs Fixed:

- The default value for `SEC_DEFAULT_SESSION_DURATION` was effectively 3600 in Condor version 7.5.0. This produced longer than desired cached sessions for short-lived tools such as *condor_status*. It also produced shorter than possibly desired cached sessions for long-lived daemons such as *condor_startd*. The default has been restored to what it was before Condor version 7.5.0, with the exception of *condor_submit*, which has been changed from 1 hour to 60 seconds. For command line tools, the default is 60 seconds, and for daemons it is 1 day.
- `SEC_<access-level>_SESSION_DURATION` previously did not support integer expressions, but it did not detect invalid input, so the use of an expression could produce unexpected results. Now, like other integer configuration variables, a constant expression can be used and input is fully validated.

- The configuration variable `LOCAL_CONFIG_DIR` is no longer ignored if defined in a local configuration file.
- Removed the incorrect issuing of the following Condor version 7.5.0 warning to the *condor_starter*'s log, even when the outdated, and no longer used configuration variable `EXECUTE_LOGIN_IS_DEDICATED` was not defined.

```
WARNING: EXECUTE_LOGIN_IS_DEDICATED is deprecated.
Please use DEDICATED_EXECUTE_ACCOUNT_REGEXP instead.
```

Known Bugs:

- None.

Additions and Changes to the Manual:

- None.

Version 7.5.0

Release Notes:

- All bug fixes and features which are in 7.4.1 are in this 7.5.0 release.

New Features:

- Added the new daemon *condor_shared_port* for Unix platforms (except for HPUX). It allows Condor daemons to share a single network port. This makes opening access to Condor through a firewall easier and safer. It also increases the scalability of a submit node by decreasing port usage. See section 3.3.36 for more information.
- Improved CCB's handling of rude NAT/firewalls that silently drop TCP connections.
- Simplified the publication of daemon addresses. `PublicNetworkIpAddr` and `PrivateNetworkIpAddr` have been removed. `MyAddress` contains both public and private addresses. For now, `<Subsys>IpAddr` contains the same information. In a future release, the latter may be removed.
- Changes to `TCP_FORWARDING_HOST`, `PRIVATE_NETWORK_ADDRESS`, and `PRIVATE_NETWORK_NAME` can now be made without requiring a full restart. It may take up to one *condor_collector* update interval for the changes to become visible.

- Network compatibility with Condor prior to 6.3.3 is no longer supported unless SEC_CLIENT_NEGOTIATION is set to NEVER. This change removes the risk of communication errors causing performance problems resulting from automatic fall-back to the old protocol.
- For efficiency, authentication between the *condor_shadow* and *condor_schedd* daemons is now able to be cached and reused in more cases. Previously, authentication for updating job information was only cached if read access was configured to require authentication.
- *condor_config_val* will now report the default value for configuration variables that are not set in the configuration files.
- The *condor_gridmanager* now uses a single status call to obtain the status of all CREAM grid universe jobs from the remote server.
- The *condor_gridmanager* will now retry CREAM commands that time out.
- Forwarding a renewed proxy for CREAM grid universe jobs to the remote server is now much more efficient.

Configuration Variable and ClassAd Attribute Additions and Changes:

- Removed the configuration variable COLLECTOR_SOCKET_CACHE_SIZE. Configuration of this parameter used to be mandatory to enable TCP updates to the *condor_collector*. Now no special configuration of the *condor_collector* is required to allow TCP updates, but it is important to ensure that there are sufficient file descriptors for efficient operation. See section 3.7.6 for more information.
- The new configuration variable USE_SHARED_PORT is a boolean value that specifies whether a Condor process should rely on the *condor_shared_port* daemon for receiving incoming connections. Write access to DAEMON_SOCKET_DIR is required for this to take effect. The default is False. If set to True, SHARED_PORT should be added to DAEMON_LIST. See section 3.3.36 for more information.
- Added the new configuration variable CCB_HEARTBEAT_INTERVAL. It is the maximum number of seconds of silence on a daemon's connection to the CCB server after which it will ping the server to verify that the connection still works. The default value is 1200 (20 minutes). This feature serves to both speed up detection of dead connections and to generate a guaranteed minimum frequency of activity to attempt to prevent the connection from being dropped.

Bugs Fixed:

- Fixed problem with a ClassAd debug function, so it now properly emits debug information for ClassAd IfThenElse clauses.

Known Bugs:

- None.

Additions and Changes to the Manual:

- None.

8.5 Stable Release Series 7.4

This is a stable release series of Condor. As usual, only bug fixes (and potentially, ports to new platforms) will be provided in future 7.4.x releases. New features will be added in the 7.5.x development series.

The details of each version are described below.

Version 7.4.5

Release Notes:

- Condor version 7.4.5 not yet released.

New Features:

- *condor_dagman* now prints a message in the `dagman.out` file whenever it truncates a node job user log file.
- *condor_dagman* now prints additional diagnostic information in the case of certain log file errors.

Configuration Variable and ClassAd Attribute Additions and Changes:

- None.

Bugs Fixed:

- A network disconnect between the submit machine and execute machine during the transfer of output files caused the *condor_starter* daemon to immediately give up, rather than waiting for the *condor_shadow* to reconnect. This problem was introduced in Condor version 7.4.4.
- If *condor_ssh_to_job* attempted to connect to a job while the job's input files were being transferred, this caused the file transfer to fail, which resulted in the job returning to the idle state in the queue.

- In privsep mode, the transfer of output failed if a job's execute directory contained symbolic links to non-existent paths.

Known Bugs:

- None.

Additions and Changes to the Manual:

- None.

Version 7.4.4

Release Notes:

- Condor version 7.4.4 released on October 18, 2010.
- *Security Item*: This release fixes a bug in which Amazon EC2 jobs (jobs with **universe = grid** and **grid_resource = amazon**) that use the **amazon_keypair_file** command may expose the private SSH key to other users. The created file had insecure permissions, allowing other users on the submit host to read the file. Any other user who could see the file could learn about these EC2 jobs using *condor_q*, and the other user could then connect to them with the private SSH key.

  To work around the bug without installing this release, do one or both of the following:

  - Do not use the submit description file command **amazon_keypair_file**.
  - Ensure that the directory holding the private SSH key has suitably restrictive permissions, such that other users cannot read files inside the directory.
- Condor can now be built on Mac OS X 10.6.
- The *condor_master* shutdown program, which is configured via the `MASTER_SHUTDOWN_<Name>` configuration variable, is now run with root (Unix) or administrator (Windows) privileges. The adminstrator must ensure that this cannot be used in such a way as to violate system integrity.

New Features:

- **load_profile** is now supported by the Unix version of *condor_submit* when submitting jobs to Windows. Previously, this command was only supported by the Windows version of *condor_submit*.
- Added an example Mac OS X launchd configuration file for starting Condor.

Configuration Variable and ClassAd Attribute Additions and Changes:

- None.

Bugs Fixed:

- Fixed bad behavior in *condor_quill* where, under certain error conditions, many copies of the `schedd_sql.log` file would be inserted into the database, filling up the disk volume used by the database. As a consequence of this bug fix, the `LogBody` column for each row in the `Error_SqlLogs` table will be empty. Please consult the *condor_quill* daemon log file for the error instead.
- Fixed a bug with how the **standard** universe remote system call `getrlimit()` functioned. Under certain conditions with 32-bit and 64-bit **standard** universe binaries, `getrlimit()` would erroneously report 2147483647 bytes as a limit, when `RLIM_INFINITY` should have been the correct response.
- Fixed a misleading error message issued by *condor_run*, which stated

  ```
  The DAGMan job was aborted by the user.
  ```

  when the job submitted by *condor_run* was aborted by the user. It now states

  ```
  The job was aborted by the user.
  ```

- When the *condor_startd* daemon is running with an execute directory on a very large file system, with more than 32 bits worth of free blocks on a 32-bit system, it would incorrectly report 0 free bytes. This has been fixed.
- For spooled jobs, input files were sometimes transferred twice from the submit machine to the execute machine. This happened if the input files were specified without any path information, as with a file name with no directory specified. This problem has existed since before Condor version 7.0.0.
- The configuration variable `NETWORK_INTERFACE` did not work in some situations, because of Condor's attempts to automatically rewrite published addresses to match the IP address of the network interface used to make the publication.
- Fixed a bug in which the default unit of configuration variable `STARTD_CRON_TEST_PERIOD` should have been seconds, but instead was `Undefined`.
- Fixed a bug in which *condor_submit* checked for bad *condor_schedd* cron arguments incorrectly within a submit description file. Now *condor_submit* will detect the problem and print out an error message.
- With some versions of *ssh*, *condor_ssh_to_job* failed if the `SHELL` environment variable was set to */bin/csh*.

- Submission of **vm** universe jobs via Globus was not possible, because the Globus Condor jobmanager explicitly set the input, output, and error files to `/dev/null`, and *condor_submit* refused any setting of these files for **vm** universe jobs. Now, `/dev/null` is an allowed setting for the input, output, and error files for **vm** universe jobs.
- Fixed a bug that caused a **vm** universe job's output files to be incorrectly transferred back to the submit machine, when the submit description file command **vm_no_output_vm** was set to `false`, indicating that no files should be transferred.
- String literals within `$$([])` expressions within a submit description file failed to be evaluated and resulted in the job going on hold. This problem has existed since before Condor 7.0.0.
- *condor_preen* was not able to clean up files in the `EXECUTE` directory when in privsep mode.
- A problem was fixed that could cause a Condor daemon that connects to other daemons via CCB to permanently run out of space for more registered sockets until restarted. This problem appeared in the logs as the following message:

  ```
  file descriptor safety level exceeded
  ```

- Fixed a problem that could cause the *condor_collector* to crash when receiving updated matchmaking information for offline ClassAds that do not exist.
- *condor_ssh_to_job* did not work when `SEC_DEFAULT_NEGOTIATION` was set to `OPTIONAL`.
- The **vm** universe now works properly on machines that have Condor's Privilege Separation mechanism enabled.
- *condor_submit* no longer pads a **vm** universe job's disk usage estimation by 100MB.
- Fixed a bug with the `vm_cdrom_files` submit file command, that caused VMware **vm** universe jobs to fail if the virtual machine already had a CD-ROM image associated with it.
- Configuration variables `SOAP_SSL_CA_DIR` and `SOAP_SSL_CA_FILE` are now properly used when authenticating with Amazon EC2 servers.
- Fix a bug with the `<subsys>_LOCK` configuration variable. It could let daemons writing to the same daemon log overwrite each other's entries and cause daemons to exit when the log is rotated.
- Fixed a bug that caused nordugrid jobs to fail if the **grid_resource** attribute included a port as part of the server host name.
- Fixed a confusing error message mentioning `LocalUserLog::logStarterError()` when the *condor_starter* fails to communicate with the *condor_shadow* before the job has started.

- Fixed the event log and shadow log for standard universe jobs to identify the checkpoint server on which a job might have failed to store its checkpoint or from which it might have failed to restore its checkpoint.
- Fixed a bug in the *condor_gridmanager* that could cause it to crash while handling grid-type cream jobs.
- Improved the *condor_gridmanager*'s handling of grid-type cream jobs that are held or removed by the user. Canceling the cream job is much less likely to fail and jobs can no longer get stuck in the cream state of CANCELED.
- Fixed the web server feature controlled by `ENABLE_WEB_SERVER` . Previously, all HTTP GET requests would fail on non-linux Unix machines.

Known Bugs:

- None.

Additions and Changes to the Manual:

- The Windows platform installation instructions have been updated.
- Section 2.5.4 on Condor's File Transfer Mechanism has been revised and updated.
- Section 4.1.4, providing examples of utilizing ClassAd expressions within the **-constraint** option of *condor_q* or *condor_status* commands has been expanded to clarify both Unix and Windows platform specifics.

Version 7.4.3

Release Notes:

- Condor version 7.4.3 released on August 16, 2010.

New Features:

- None.

Configuration Variable and ClassAd Attribute Additions and Changes:

- The new configuration variable `ENABLE_CHIRP` defaults to `True`. An administrator may set it to `False`, which disables Chirp remote file access from execute machines.

- The new configuration variable `ENABLE_ADDRESS_REWRITING` defaults to `True`. It may be set to `False` to disable Condor's dynamic algorithm for choosing which IP address to publish in multi-homed environments. The dynamic algorithm chooses the IP address associated with the network interface used to make the publication, for example, the network interface used to communicate with the *condor_collector*.
- Configuration variable `VM_BRIDGE_SCRIPT` has been removed and is no longer valid.
- The new configuration variable `VM_NETWORKING_BRIDGE_INTERFACE` specifies the networking interface that Xen or KVM **vm** universe jobs will use. See section 3.3.29 for documentation.
- Allowed the configuration file entries `GSI_DAEMON_TRUSTED_CA_DIR` and `GSI_DAEMON_DIRECTORY` to be set with environment variables, as the rest of Condor configuration variables can be.

Bugs Fixed:

- When using file transfer semantics, if the job exited in such a manner so as to not produce all output files specified in **transfer_output_files**, then which files were transferred was potentially incorrect. Now, all existing files are transferred back, and the files which are not able to be transferred back due to non-existence appear as zero length files. An example of when this occurred would be the job dumping core and then being placed on hold.
- Fetch work hooks to prepare are now invoked as the `condor` user, instead of as the job user.
- Improved the file extension detection on Windows platforms.
- *condor_wait* could occasionally get stuck in an infinite loop, if it missed the execution event of the job it is waiting for. This is now fixed.
- Fixed a bug within the *condor_startd* cron capabilities, that only occurred on Windows platforms. *condor_startd* cron scripts failed to run if an initial directory was left unspecified.
- Fixed a bug in which a job would be incorrectly placed on hold, with a confusing error message that appeared similar to

  ```
  Condor failed to start your job 9090.-1 because job attribute Args contains $$(VirtualMachineID).
  ```

  This occurred if the submit command **copy_to_spool** was `true`, the submit description file for the job contained $$ macros, and *condor_preen* ran after the job was submitted and before it started.
- Added the jobs_vertical_history table to the list of tables that *condor_quill* periodically re-indexes.
- Fixed bug in *condor_preen* in which it would delete *condor_startd* daemon history files.

- Fixed a bug where if a user job using file transfer with **transfer_output_files** and **when_to_transfer_output** is set to `ON_EXIT_OR_EVICT` fails to produce all of the specified files and exit, as when core dumping due to a fault, then the stdout, stderr, and core file of the job were not transferred back to the submitting machine.
- Fixed numerous, small, rare memory leaks.
- Fixed a bug in which processor affinity settings were ignored if privilege separation was enabled.
- Network connections for Condor file transfers were ignoring private network settings. The connection from the execute node to the submit node always attempted to use the public network address of the submit machine.
- The configuration variable `TCP_FORWARDING_HOST` did not work in some situations because of Condor's attempts to automatically rewrite published addresses to match the IP address of the network interface used to make the publication.
- A single job could match multiple offline slots in a single negotiation cycle. This problem could cause *condor_rooster* to wake up too many offline machines for the number of jobs available to run on them. The fix for this problem requires updating both the *condor_negotiator* and the *condor_schedd*.
- Fixed a problem that caused the *condor_startd* daemon to crash in some cases when `STARTD_SENDS_ALIVES` was `True`. This setting is `False` by default.
- Fixed a problem where the *condor_kbdd* has a chance of entering an infinite loop on platforms that use X-Windows. Microsoft Windows and Mac OS X platforms were not affected. This bug is present in all earlier 7.4.x Condor releases.
- The default path to *sftp-server* has been improved, so that *condor_ssh_to_job* can use *sftp* out-of-the-box on RedHat Enterprise Linux 5 platforms.
- A *nordugrid_gahp* binary built on RedHat Enterprise Linux 3 no longer crashes when run on a RedHat Enterprise Linux 4 or Scientific Linux 4 machine.
- Fixed a bug in *condor_rm* that caused it to misinterpret user names that begin with a digit, such as `4abc`. It incorrectly used them as cluster numbers.
- Fixed a bug that caused the *condor_startd* to invoke the "power_state" plug-in as the condor user. This caused hibernation to fail because power_state requires root privileges to function properly.
- Fixed a bug that could cause the *condor_schedd* to crash if there were any idle scheduler universe jobs when files were staged into the *condor_schedd* for a new job.
- Fixed a bug in the *nordugrid_gahp* that could cause it to exit when connecting to a misconfigured LDAP server.
- Fixed a bug that prevented the log file defined with the configuration variable `NEGOTIATOR_MATCH_LOG` from rotating. See section 3.3.4 for the definition of this variable.

- Fixed a bug that caused *startd_cron* jobs to fail on Windows. This bug is present in all earlier 7.4.x Condor releases.
- The submit description file command **vm_cdrom_files** now works properly with Windows execute machines. Previously, creation of the ISO file would fail, causing job execution to be aborted.
- Fixed a bug that caused the *condor_startd* to invoke the *power_state* plug-in as the condor user. This caused hibernation to fail, because *power_state* requires root privileges to function properly.

Known Bugs:

- None.

Additions and Changes to the Manual:

- Searching the PDF version of the manual for items containing underscore characters, such as many configuration variable names, now works correctly.
- The new subsection 4.1.3 provides examples of evaluation results when using the operators `==`, `=?=`, `!=`, and `=!=`.
- Section 2.11 with specifics on **vm** universe jobs has been updated to contain more details about both checkpoints and **vm** universe jobs in general.

Version 7.4.2

Release Notes:

- Condor version 7.4.2 released on April 6, 2010.

New Features:

- None.

Configuration Variable and ClassAd Attribute Additions and Changes:

- When `WANT_SUSPEND` is defined and evaluates to anything other than the value `True`, it is now utilized as if it were `False`. If `WANT_SUSPEND` is not explicitly defined, the *condor_startd* exits with an error message. Previously, if `Undefined`, it was treated as an error, which caused the *condor_startd* to exit with an error message.

Bugs Fixed:

- Fixed a bug in which the *condor_schedd* would sometimes negotiate for and try to run more jobs than specified by `MAX_RUNNING_JOBS`. Once the jobs started running, it would then kill them off to get back below the limit. This was more likely to happen with slow preemption caused by `MaxJobRetirementTime` or by a large timeout imposed by `KILL`. This problem has existed since before Condor 6.5. When this problem happened, the following message appeared in the *condor_schedd* log:

  ```
  Preempting X jobs due to MAX_JOBS_RUNNING change
  ```

- Fixed a problem that caused *condor_ssh_to_job* to fail to connect to a job running on a slot with multiple '@' signs in its name. This bug has existed since the introduction of *condor_ssh_to_job* in 7.3.2.

- In all previous versions of Condor, *condor_status* refused to accept **-long**, **-xml**, and **-format** when followed by an argument such as **-master** that specified which type of daemon to look at. The order of the arguments had to be reversed or it would produce a message such as the following:

  ```
  Error:  arg 4 (-master) contradicts arg 1 (-format)
  ```

- Fixed a bug which caused the *condor_master* to crash if `VIEW_SERVER` was included in `DAEMON_LIST` and `CONDOR_VIEW_HOST` was unset.

- Fixed a bug that caused configuration parameter `LOCAL_CONFIG_DIR` to be ignored if it was set in a local configuration file, as opposed to the top-level configuration file.

- Fixed a bug that could cause the *condor_schedd* to behave incorrectly when reading an invalid job queue log on startup.

- Fixed a bug that could corrupt the job queue log if the *condor_schedd* daemon's attempt to compact it fails.

- Fixed a problem that in rare cases caused the *condor_schedd* to crash shortly after the *condor_gridmanager* exited. This bug has existed since before Condor version 6.8.

- Fixed a problem that was resulting in messages such as the following:

  ```
  ERROR: receiving new UDP message but found a long message still waiting
  to be closed (consumed=0). Closing it now.
  ```

- The file extension specified to *condor_fetch_log* can no longer contain a path delimiter.

- When in graceful shutdown mode, the *condor_schedd* was sometimes starting idle scheduler universe jobs. With a large enough number of scheduler universe jobs, this could lead to a cycle of stopping and restarting jobs until the graceful shutdown time expired.

- Fixed multiple bugs that prevented Condor from building on or running correctly on OpenSolaris X86/64 version 2009.06.
- Fixed a bug which caused the *condor_startd* to incorrectly count the number of processors on some machines with Hyper-threading enabled. This bug was introduced in Condor version 7.3.2, and exists in 7.4.0 and 7.4.1.
- Fixed a problem with GSI authentication in Condor that would cause daemons to consume more and more memory over time. The biggest source of trouble was introduced in Condor version 7.3.2. However, a smaller memory leak that existed in all previous versions of Condor has also been fixed.
- Fixed a bug where if *condor_compile* is invoked in a manner such as:

  ```
  condor_compile gcc -print-prog-name=ld
  ```

  an error would be emitted, and *condor_compile* would exit with a bad exit code.
- The sort based on *condor_status* output accidentally changed in Condor version 7.3, so that the output was based on the slot name first, then machine name. The behavior is now restored to the original sorting: first on machine name, then slot name.
- If one machine running a parallel job crashed, and job leases are enabled (which they are by default), the job would not exit until the job lease duration expired. As the *condor_starter* will not get respawned, there is no need to wait. Many sites set long job lease durations, to prevent jobs from being killed when the machine running the *condor_schedd* daemon reboots. Now, if one node goes away, the whole computation is shut down immediately.
- Fixed the verbosity level of some *condor_dagman* messages written to the `dagman.out` file.
- Fixed a bug introduced in Condor version 7.3.2 that resulted in messages such as the following even in cases where no problem in communicating with the *condor_collector* had been encountered:

  ```
  Collector <X> is still being avoided if an alternative succeeds.
  ```

  This problem was believed to be fixed in Condor 7.4.1, but some cases of the problem remained in that version.
- Fixed a bug from Condor version 6.1.14, that resulted in the *condor_schedd* performing the operation scheduled via `WALL_CLOCK_CKPT_INTERVAL` at the specified frequency (default time of 1 hour), multiplied by the number of times the *condor_schedd* daemon had been reconfigured during its lifetime. This could lead to degraded performance, especially prior to Condor version 7.4.1, when this operation was more disk-intensive.
- 32-bit Linux versions of Condor running in a 64-bit environment would sometimes not detect the existence of some processes and sometimes wrongly detect that a tracked process belonged to root when it actually belonged to some other user. This could lead to failure to run jobs or failure to properly monitor and clean up after them. When the wrong process ownership problem happened, the following message appeared in the *condor_master* and/or *condor_procd* logs:

```
ProcAPI: fstat failed in /proc! (errno=75)
```

If *condor_procd* failed to detect the existence of its own parent process, it would exit with the following message in its log:

```
ERROR: master has exited
```

- Fixed a problem in the *condor_job_router* daemon, introduced in Condor version 7.2.2, that could cause the daemon to crash when failing to carry out the change of state dictated by a job's periodic policy expressions, for example, the failure to put a job on hold when `periodic_hold` becomes `True`.
- Fixed a bug introduced in Condor 7.3.2 that caused Grid Monitor jobs to receive a full X.509 proxy. Now, it always receives a limited proxy, which was the previous behavior.
- Fixed a bug that could cause the nordugrid_gahp to crash.
- Fixed a problem introduced in 7.4.0 that could cause two *condor_schedd* daemons with a match to the same slot to both fail to claim it, rather than letting the first one to claim it succeed. This sort of situation can happen when the *condor_negotiator* has a stale view of the pool, either because the gap between negotiation cycles is configured to be shorter than usual, or because updates from the *condor_startd* to the *condor_collector* are not reliably delivered and processed.
- The *condor_kbdd* is no longer ignored by the *condor_startd* when the configuration variable `CONSOLE_DEVICES` is defined.
- When using the **-d** command line argument with a daemon, the values of `LOG`, `SPOOL`, and `EXECUTE` no longer change every time a *condor_reconfig* command is received.

Known Bugs:

- The *condor_kbdd* has a chance of entering an infinite loop on platforms that use X-Windows. Microsoft Windows and Mac OS X are not affected. Removing KBDD from `DAEMON_LIST` is a workaround, although this impairs Condor's ability to detect console usage. This bug is fixed in Condor version 7.4.3.

Additions and Changes to the Manual:

- Descriptions of all the commands that may be placed into a submit description file are now located within the *condor_submit* manual page, instead of within Chapter 2, the Users' Manual.
- An initial, but not yet complete set of configuration variables that require a restart when changed, is listed in section 3.3.1. Using *condor_reconfig* to change these variables' values is not sufficient.

Version 7.4.1

Release Notes:

- *Security Item*: A flaw was found that could allow a user who already is authorized to submit jobs into Condor, to queue a job under the guise of a different user. In this way, someone who has access to a Condor submission service and is allowed to submit jobs into Condor could gain access to another non-root or non-administrator account on the system. This flaw was discovered during the development process; no incidents have been reported. Details of the problem will be made available on Feb 1st, 2010.
- The default value of `JOB_ROUTER_NAME` has changed from an empty string to `jobrouter` in order to address problems caused by the previous default. Without special handling, this means that jobs being managed by *condor_job_router* before upgrading will not be adopted by the new version of *condor_job_router* if the default `JOB_ROUTER_NAME` was being used. To correct this, follow the instructions given in the description of `JOB_ROUTER_NAME` on page 240.

New Features:

- Condor allows submit files to specify an **IwdFlushNFSCache** expression, to control whether or not Condor tries to flush the NFS cache for a job's initial working directory on job completion.
- The new **-attributes** option to *condor_status* explicitly specifies the attributes to be listed when using the **-xml** or **-long** options.

Configuration Variable and ClassAd Attribute Additions and Changes:

- New VOMS attributes have been introduced into the job ad to keep them separate from the X509UserProxySubjectName.
- The default for `JOB_ROUTER_NAME` has changed from an empty string to `jobrouter`. See the release notes for more information about upgrading from an old version.
- The configuration variable `TCP_FORWARDING_HOST` has existed in Condor since version 7.0.0, but was not documented. See section 3.3.6 for documentation.
- The new configuration variable `STARTD_PER_JOB_HISTORY_DIR` allows ClassAds of completed jobs to be stored in a directory separate from the existing one specified with `PER_JOB_HISTORY_DIR`.

Bugs Fixed:

- Condor no longer creates the job sandbox in its `SPOOL` directory if it is not needed.

- Fixed a problem introduced in Condor version 7.4.0 that caused GSI authentication between Condor processes to fail with using a non-legacy format X.509 proxy.
- Fixed a problem with CCB under Windows platforms that has existed since Condor version 7.3.0. This problem caused CCB-enabled daemons to become unresponsive after the exit of a child process.
- Improved the handling of previously-submitted gt2 grid jobs upon release from hold, when there is no Globus job manager for the job running on the remote resource.
- Fixed a problem with job leases for jobs that use a *condor_shadow*. Previously, while these jobs were running, lease renewals from the submitter would not be noticed, and the job would be aborted when the original lease expired.
- Fixed a bug that only allowed approximately 50 splices to be included into a DAG input file. There is now no limit to the number of splices one may include into a DAG input file except, of course, for the implicit memory allocation limit of the *condor_dagman* process.
- Removed attempted limiting of swap space via *ulimit -v* using the `VirtualMemory` machine ClassAd attribute in the script `condor_limits_wrapper.sh`.
- Fixed a bug that caused `ALLOW_CONFIG` and `HOSTALLOW_CONFIG`, as well as the corresponding `DENY` configuration variables to incorrectly handle a setting consisting of a single `*` or the equivalent `*/*`. This also fixes a bug that caused incorrect merging of `ALLOW` and `HOSTALLOW` settings when one, but not both, consisted of a single `*` or the equivalent `*/*`. These bugs have existed since before Condor version 6.8.
- Fixed a bug introduced in Condor version 7.3.0 that could cause Condor daemons to crash when reading malformed network addresses.
- Removed a check for root ownership of a script specified by the configuration variable `VM_SCRIPT`.
- Fixed a bug in writing the header of the file identified by the configuration variable `EVENT_LOG`.
- Fixed a bug that could cause the *condor_startd* to segfault on shutdown when using dynamic slots.
- Fixed a problem introduced in Condor version 7.3.2 that changed the behavior of an undocumented method for selecting attributes to be displayed in *condor_q* **-xml**. Prior to this bug, the following command would produce XML output with the attributes `A` and `B`, plus a few other attributes that were always shown.

  ```
  condor_q -xml -format "%s" A+B
  ```

  In Condor versions 7.3.2 and 7.4.0, this same command produced an empty XML ClassAd. The workaround was to use multiple **-format** options, each listing just one desired attribute, rather than a single one with an expression of all desired attributes. Although this is now fixed, the more straightforward way to select attributes since Condor version 7.3.2 is to use the **-attributes** option.

- Fixed a bug introduced in Condor version 7.3.2 that resulted in messages such as the following even in cases where no problem in communicating with the *condor_collector* had been encountered:

  ```
  Collector <X> is still being avoided if an alternative succeeds.
  ```

- Fixed a bug that has been in the *condor_startd* since before Condor version 6.8. If the *condor_startd* ever failed to send signals to the *condor_starter* process, it could fail to properly clean up the machine ClassAd, leaving attributes from `STARTD_JOB_EXPRS` in the ClassAd but not making them visible in *condor_status* queries. One possible problem resulting from this could be matches being made by the *condor_negotiator* that are then rejected by the *condor_startd*. Repeated messages such as the following would then result in the *condor_startd* log:

  ```
  slot1: Request to claim resource refused.
  ```

- Fixed a problem that resulted in the following message in the *condor_startd* log:

  ```
  Timer -1 not found
  ```

- Fixed a problem in which security sessions were not cached correctly when using CCB. This resulted in re-authentication in some cases where a cached security session could have been used.
- Fixed multiple problems with the handling of VOMS attributes in GSI proxies.
- Fixed a bug that caused *condor_dagman* to hang when running a DAG with POST scripts, if the global event log is turned on.
- Improved how the private network address is published when using the configuration variables `PRIVATE_NETWORK_NAME` and `PRIVATE_NETWORK_INTERFACE`. In some cases, this information was not being used, and therefore connections were made to the public address when they could have been made to the private address.
- Fixed a bug exhibited under Windows XP, where using `USE_VISIBLE_DESKTOP` would cause strange behavior after a job completed.
- CCB now works with `TCP_FORWARDING_HOST`. Previously, the reverse connection was made to the private address rather than to the host defined by `TCP_FORWARDING_HOST`.
- Removed a bad optimization that caused some information about job execution to be lost during job completion or removal, if a history file was not configured.
- Condor now checks whether the configuration variable `GRIDFTP_URL_BASE` is set before submitting cream grid jobs, as that variable is required for cream jobs to function properly. If the variable is not set, cream jobs are put on hold with an appropriate message.
- Fixed a bug that allowed running virtual machines to be leaked if the *condor_startd* crashed.

- Fixed a bug in *cream_gahp* which could cause crashes when there were more than 500 cream jobs queued.
- Improved recovery when Condor crashes during the submission of a cream grid job. Before, affected jobs would remain in `REGISTERED` state on the cream server, but never run.
- Improved the `HoldReason` message when cream grid jobs are held by the *condor_gridmanager*.
- When naming a resource for a cream grid job, Condor now properly recognizes the format used by the standard cream client UI: `https://foo.edu:8443/cream-pbs-cream_queue`.
- The configuration variable `SOAP_SSL_CA_FILE` is now consulted in addition to `SOAP_SSL_CA_DIR` when authenticating an https proxy for Amazon EC2, when `AMAZON_HTTP_PROXY` is defined.
- Previously, if *condor_rm* and friends were given both a constraint and a user name or cluster id, they would act on all jobs matching the constraint and all jobs associated with the user or cluster. Now, this combination of arguments results in an error.
- Failure to purge a cream grid universe job from the remote server because it was previously purged no longer results in the job being held.
- The *condor_gridmanager* now recognizes VOMS attributes in X.509 proxies and will handle them appropriately. For example, it recognizes that two proxies with the same identity but different VOMS attributes may be mapped to different accounts on a remote machine.
- Fixed a bug in *condor_dagman*, introduced in 7.3.2, that will cause *condor_dagman* running on Windows to hang on any DAG using more than one log file for the node jobs.
- Fixed a bug in *condor_dagman*, introduced in 7.3.2, that could cause *condor_dagman* to fail on a DAG using node job log files on multiple devices, if log files on different devices happened to have the same inode number.
- Fixed a bug that caused the *condor_schedd* daemon to segfault when spooling more than 9 files.
- Fixed a bug that caused the *condor_startd* daemon to crash on Debian Stable.
- Fixed keyboard activity detection on the Windows XP platform.
- Fixed a bug in the *condor_had* daemon that caused it to not start the controlled daemon if CCB was enabled.

Known Bugs:

- The *condor_kbdd* has a chance of entering an infinite loop on platforms that use X-Windows. Microsoft Windows and Mac OS X are not affected. Removing KBDD from `DAEMON_LIST` is a workaround, although this impairs Condor's ability to detect console usage. This bug is fixed in Condor version 7.4.3.

- *condor_dagman* may fail on Windows if the set of node job log file names includes multiple paths that are hard links (not symbolic links) to the same file.
- *condor_dagman* PRE and POST script arguments (and the names of the scripts themselves) cannot contain spaces.
- *condor_dagman* VARS values cannot contain single quotes.

Additions and Changes to the Manual:

- Added documentation about how to include spaces (and other special characters) in *condor_dagman* VARS values.

Version 7.4.0

Release Notes:

- The default configuration file within the release now uses `ALLOW`/`DENY` in place of `HOSTALLOW`/`HOSTDENY` for security related settings. We recommend making this same change throughout all configuration files. That way, a policy that depends on the default policy should continue to work as it did before. The behavior of these configuration variables remains unchanged. The `ALLOW`/`DENY` lists are added to the `HOSTALLOW`/`HOSTDENY` lists to form the authorization policy. Both styles support the same syntax. This change permits an anticipated phasing out of the `HOSTALLOW`/`HOSTDENY` configuration variables, in order to simplify configuration.
- As of Condor version 7.3.2, *condor_q* **-xml** output no longer begins with the non-XML consisting of two blank lines followed by a line of the following form:

  ```
  -- Submitter: schedd-name : <IP> : hostname
  ```

- All *Stork* data placement is now supported by the Stork project at the LSU Center for Computation and Technology (http://www.cct.lsu.edu/www.cct.lsu.edu). Please see the home page of the Stork project at http://www.cct.lsu.edu/ kosar/stork/index.php for details and software.

New Features:

- Condor is now integrated with the Hadoop Distributed File System (HDFS). See documentation in section 3.13.2 and section 3.3.23.
- *condor_q* using the options **-analyze** and **-better-analyze** now provide analysis for scheduler and local universe jobs. Specifically, the `START_SCHEDULER_UNIVERSE` and `START_LOCAL_UNIVERSE` expressions are checked.

- Added the ClassAd attributes `TotalLocalRunningJobs`, `TotalLocalIdleJobs`, `TotalSchedulerRunningJobs`, and `TotalSchedulerIdleJobs` to the published ClassAd for the *condor_schedd*. This means that *condor_q* **-analyze** can still give helpful information about why local or scheduler universe jobs are idle when the configuration variables `START_LOCAL_UNIVERSE` or `START_SCHEDULER_UNIVERSE` refer to these attributes. These attributes were already present internally within the *condor_schedd* daemon, just not published.
- The *condor_vm-gahp* now supports KVM and links with libvirt, rather than calling virsh command-line tools.
- Greatly improved the *condor_gridmanager*'s scalability when handling many grid type gt2 grid universe jobs. Improvements include more quickly processing updated X.509 certificates, not checking jobs for status updates if they have not been submitted to the remote site, and eliminating unnecessary updates to the *condor_schedd* daemon.
- Latency in the submission and cleaning up of Condor-C jobs has been improved by changing the default value of `C_GAHP_CONTACT_SCHEDD_DELAY` from 20 to 5.
- The `eval()` ClassAd function added in Condor version 7.3.2 is now also understood by the *condor_job_router* and *condor_q* using the **-better-analyze** option.
- The submit command **run_as_owner** is now supported for Unix platforms. Previously, it was only supported on Windows platforms.
- When setting `AMAZON_HTTP_PROXY`, a username and password can now be given as part of the proxy URL. The value of `SOAP_SSL_CA_DIR` is now consulted when authenticating an https proxy for Amazon EC2, when `AMAZON_HTTP_PROXY` is defined.
- The *condor_collector* daemon now advertises to itself, and will appear in the output of *condor_status* **-collector**.
- Optimizations in core Condor systems should provide minor speed improvements.
- Updated the maximum log size to the maximum operating system's file size.

Configuration Variable and ClassAd Attribute Additions and Changes:

- The undocumented configuration variable `TOOLS_PROVIDE_OLD_MESSAGES` is no longer recognized by Condor.
- The new configuration variable `SCHEDD_JOB_QUEUE_LOG_FLUSH_DELAY` sets an upper bound in seconds on how long it takes for changes to the job ClassAd to be visible to the Condor Job Router and to Quill. The default value is 5 seconds. Previously, there was no upper limit. Typically, other activity in the job queue, such as jobs being submitted or completed would cause buffered data to be flushed to disk, such that the effective upper bound was a function of how busy the job queue was.
- The default configuration file now uses `ALLOW`/`DENY` in place of `HOSTALLOW`/`HOSTDENY`. See the release notes above for more information.

- The default value for MAX_JOBS_RUNNING has changed. Previously, it was 200. Now it is defined by an expression that depends on the total amount of memory and the operating system. The default expression requires 1MByte of RAM per running job, on the submit machine. In some environments and configurations, this is overly generous and can be cut by as much as 50%. Under Windows, the number of running jobs is still capped at 200. A 64-bit version of Windows is recommended in order to raise the value above the default. Under Unix, the maximum default is now 10,000. To scale higher, we recommend that the system ephemeral port range is extended such that there are at least 2.1 ports per running job.
- The default value of RESERVED_SWAP has changed to the value 0, which disables the *condor_schedd* daemon's check for sufficient swap space before starting more jobs. The new expression defined with MAX_JOBS_RUNNING has a more appropriate memory check, so the configuration variable RESERVED_SWAP will no longer be used in the near future. For cases where RESERVED_SWAP is not set to 0, the default value of SHADOW_SIZE_ESTIMATE has changed to 800 Kbytes. Previously, it was 200 if not set, but it was set to 1800 in the example configuration file.
- The default values of START_LOCAL_UNIVERSE and START_SCHEDULER_UNIVERSE have changed. Previously, these were set to True. Now, they are set using an expression that allows up to 200 local universe and 200 scheduler universe jobs to run.
- The default value of GRIDMANAGER_MAX_SUBMITTED_JOBS_PER_RESOURCE has changed from 100 to 1000.
- The default value of NEGOTIATOR_INTERVAL has changed from 300 to 60.
- The default value of ENABLE_GRID_MONITOR has been changed from False to True. This variable was assigned to True in the example configuration file, so the change in default value now matches the value set in the example configuration.
- The configuration variable VM_VERSION has been removed, as has the machine ClassAd attribute of the same name. When the virtual machine version information is needed in the machine ClassAd, the configuration variable STARTD_ATTRS can be used to add it.
- The default configuration now uses VM_BRIDGE_SCRIPT and VM_SCRIPT in place of XEN_BRIDGE_SCRIPT and XEN_SCRIPT due to the support of KVM. Submit description file commands have also been added, and they include: **kvm_disk**, **kvm_transfer_files**, and **kvm_cd_rom_device**.
- The configuration variables XEN_DEFAULT_KERNEL and XEN_DEFAULT_INITRD have been removed. Corresponding to this, the submit description file command xen_kernel = any is no longer valid.

Bugs Fixed:

- Fixed a bug that prevented parallel universe jobs from running on *condor_startd* daemons with dynamic slots enabled.

- Fixed a race condition bug in the *condor_startd* which allowed it to send Unix signals, intended for *condor_starter* processes, as root to non-Condor related processes.
- A Windows platform bug has been fixed. The bug caused a 20-second interval in which the *condor_shadow*, *condor_startd*, and *condor_starter* daemons appeared as deadlocked. The bug was visible if a job ClassAd update from the *condor_starter* caused the job's periodic hold or remove policy to become `True`.
- Fixed a bug that could cause *condor_dagman* to generate an illegal rescue DAG, if it read events incorrectly in recovery mode. *condor_dagman* now checks for events that violate DAG semantics when reading events in recovery mode, and it exits without creating a rescue DAG if it reads such an event.
- Fixed a bug that could cause *condor_dagman* to abort if it saw the combination of a terminated event and an aborted event on a node with retries.
- Changed some logged warnings in *condor_dagman* to not be printed at the default verbosity setting.
- The version compatibility checking between a `.condor.sub` file and the *condor_dagman* binary which is done at DAG startup is now much more permissive. Currently, `.condor.sub` files with Condor versions of 7.1.2 and later accepted by *condor_dagman*.
- Fixed a bug introduced with the new *condor_dagman* lazy log file evaluation code in Condor version 7.3.2. The bug sometimes caused failure when running rescue DAGs.
- Fixed a bug originating in Condor version 7.1.4. When a user submitted a job with an executable that did not have execute permission enabled, Condor was running as root, and file transfer was specified in the job, Condor failed to automatically turn on execute permission after transferring the file.
- Fixed a bug that appeared in Condor version 7.3.2. The configuration variable `COUNT_HYPERTHREAD_CPUS` was ignored and was effectively treated as `False` in all cases.
- Fixed a bug in which the Condor Job Router was not able to see matchmaking diagnostic attributes such as `LastRejMatchTime`. Therefore, when evaluating policy expressions that referred to these attributes, they were effectively treated as though `Undefined`. Quill was also not able to see these attributes.
- Fixed a bug introduced in Condor version 7.3.2 that could cause the *condor_gridmanager* to crash repeatedly on startup, if the job queue contained grid type gt2 jobs that had been previously submitted.
- Fixed two bugs introduced in Condor version 7.3.2, and related to VOMS. The first bug prevented jobs with X.509 proxies from being submitted on platforms on which Condor does not support VOMS. The second bug prevented submission of jobs with VOMS proxies, if the authenticity of the VOMS extensions could not be verified. At the same time, improved memory usage when VOMS extensions are not used.

- Fixed a bad default in the file `batch_gahp.config`, that prevented Condor from observing job state changes for grid universe jobs with a grid type of pbs or lsf.
- Fixed a bug that caused Condor-C jobs to fail if the submit description file command **transfer_executable** was set to `False`.
- Fixed a bug that caused Condor-C jobs to fail if the executable or one of the `stdin`, `stdout`, or `stderr` file names contained a comma.
- File transfer for grid type gt4 jobs requires an empty directory within `/tmp`, which the *condor_gridmanager* creates. If this directory is deleted, the *condor_gridmanager* will now recreate it.
- Fixed a bug that could cause the user job log to become corrupted on Windows platforms. This bug would manifest itself only if the same log file was specified with different paths. For example, the following submit file could have triggered this bug:

```
...
initialdir = /data/job1
log = ../JobLog
queue

initialdir = /data/job2
log = ../JobLog
queue
```

- Fixed a memory leak introduced into Condor version 7.3.2. The leak was in the *condor_collector* daemon.
- Fixed a bug introduced in Condor version 7.3.2 that resulted in the *condor_negotiator* daemon refusing to run, if the configuration variable `GROUP_QUOTA` for any group was set to 0.
- Fixed a bug that caused the `ctime` in the event log header to always be zero.
- Fixed the output of *condor_status* when used with the command-line options **-java** or **-vm**.
- Fixed a problem in the *condor_schedd* daemon introduced in 7.3.2. For *condor_schedd* daemons with lots of jobs having periodic release expressions, this bug could result in the *condor_schedd* taking a long time while evaluating periodic expressions, causing it to become unresponsive to queries and other tasks. With a job queue of 30,000 jobs, a period of unresponsiveness of an hour was observed, whereas the evaluation of periodic expressions in this same environment normally takes less than 5 seconds.
- Potential bugs and memory leaks were identified and fixed throughout Condor. The Condor Team is not aware of anyone having encountered these bugs.
- The *condor_starter* cleans up working directories in more situations. Previously during some error conditions, the working directory within `$(EXECUTE)` might be left behind.
- If the user log cannot be accessed when a local universe job starts, the job would fail and immediately be retried. Now the job is placed on hold.

- Fixed a bug in the *condor_startd* in which vacating jobs would not respect the value of `JobLeaseDuration`.
- Updated the detection of `HasVM` within the *condor_startd* to publish an update to the *condor_collector*, when the configuration variable `VM_RECHECK_INTERVAL` is specified.
- Fixed a bug in which the *condor_gridmanager* could, in rare cases, waste a small amount of memory and processor time checking for proxy files no longer being used by any active jobs.
- The setting `CREAM_GAHP` was added to the default configuration file with a value of `$(SBIN)/cream_gahp`. Existing installations desiring to submit jobs to CREAM should add this setting.
- Fixed a bug where *condor_restart* would fail on a *condor_collector* daemon configured for high availability with multiple *condor_collector* daemons.
- Fixed a bug in which multiple entries of output from the command *condor_status* **-negotiator** would be on a single line. They are now listed one per line.
- Fixed a bug in which the command *condor_submit* **-dump** would crash if multiple jobs were queued from within a single submit file.
- Fixed a bug in which a slot whose associated job disappeared could remain in the Claimed/Idle state until the claim lease expired. The slot should now promptly return to the Unclaimed/Idle state.
- Fixed a bug in which a *condor_startd* using dynamic slots could crash on shutdown or reconfiguration.

Known Bugs:

- The *condor_kbdd* has a chance of entering an infinite loop on platforms that use X-Windows. Microsoft Windows and Mac OS X are not affected. Removing KBDD from `DAEMON_LIST` is a workaround, although this impairs Condor's ability to detect console usage. This bug is fixed in Condor version 7.4.3.
- There are multiple bugs related to using VOMS attributes. In Condor version 7.4.0, VOMS support should be disabled by setting the configuration variable `USE_VOMS_ATTRIBUTES = FALSE`.
- A configuration variable of `USE_VISIBLE_DESKTOP` set to `True` will corrupt the visible desktop. This bug is present back through Condor version 7.2.4. This configuration variable did not work at all in 7.2 releases prior to 7.2.4. This bug will be fixed in Condor version 7.4.1.
- If the global event log (see section 3.3.4) is turned on, *condor_dagman* will hang when running any DAG that has POST scripts.
- *condor_dagman* will hang on Windows when running any DAG that uses more than one log file for the node jobs.

Additions and Changes to the Manual:

- See section 3.13.2 and section 3.3.23 for preliminary documentation of Condor's integration with the Hadoop Distributed File System (HDFS).

CHAPTER

NINE

Command Reference Manual (man pages)

cleanup_release

uninstall a previously installed software release installed by *install_release*

Synopsis

cleanup_release [**-help**]

cleanup_release *install-log-name*

Description

cleanup_release uninstalls a previously installed software release installed by *install_release*. The program works through the install log in reverse order, removing files as it goes. Each delete is logged in the install log to allow recovery from a crash. The install log name is provided as the *install-log-name* argument to this program.

Options

-help Display brief usage information and exit.

Exit Status

cleanup_release will exit with a status of 0 (zero) upon success, and non-zero otherwise.

See Also

install_release (on page 892).

Author

Condor Team, University of Wisconsin–Madison

Copyright

condor_advertise

Send a ClassAd to the *condor_collector* daemon

Synopsis

condor_advertise [**-help** | **-version**]

condor_advertise [**-pool** *centralmanagerhostname[:portname]*] [**-debug**] [**-tcp**] [**-multiple**] *update-command* [*classad-filename*]

Description

condor_advertise sends one or more ClassAds to the *condor_collector* daemon on the central manager machine. The required argument *update-command* says what daemon type's ClassAd is to be updated. The optional argument *classad-filename* is the file from which the ClassAd(s) should be read. If *classad-filename* is omitted or is the dash character ('-'), then the ClassAd(s) are read from standard input.

When **-multiple** is specified, multiple ClassAds may be published. Publishing many ClassAds in a single invocation of *condor_advertise* is more efficient than invoking *condor_advertise* once per ClassAd. The ClassAds are expected to be separated by one or more blank lines. When **-multiple** is not specified, blank lines are ignored (for backward compatibility). It is best not to rely on blank lines being ignored, as this may change in the future.

The *update-command* may be one of the following strings:

UPDATE_STARTD_AD

UPDATE_SCHEDD_AD

UPDATE_MASTER_AD

UPDATE_GATEWAY_AD

UPDATE_CKPT_SRVR_AD

UPDATE_NEGOTIATOR_AD

UPDATE_HAD_AD

UPDATE_AD_GENERIC

UPDATE_SUBMITTOR_AD

UPDATE_COLLECTOR_AD

UPDATE_LICENSE_AD

UPDATE_STORAGE_AD

condor_advertise can also be used to invalidate and delete ClassAds currently held by the *condor_collector* daemon. In this case the *update-command* will be one of the following strings:

INVALIDATE_STARTD_ADS

INVALIDATE_SCHEDD_ADS

INVALIDATE_MASTER_ADS

INVALIDATE_GATEWAY_ADS

INVALIDATE_CKPT_SRVR_ADS

INVALIDATE_NEGOTIATOR_ADS

INVALIDATE_HAD_ADS

INVALIDATE_ADS_GENERIC

INVALIDATE_SUBMITTOR_ADS

INVALIDATE_COLLECTOR_ADS

INVALIDATE_LICENSE_ADS

INVALIDATE_STORAGE_ADS

For any of these INVALIDATE commands, the ClassAd in the required file consists of three entries. The file contents will be similar to:

```
MyType = "Query"
TargetType = "Machine"
Requirements = Name == "condor.example.com"
```

The definition for `MyType` is always `Query`. `TargetType` is set to the `MyType` of the ad to be deleted. This `MyType` is `DaemonMaster` for the *condor_master* ClassAd, `Machine` for the *condor_startd* ClassAd, `Scheduler` for the *condor_schedd* ClassAd, and `Negotiator` for the *condor_negotiator* ClassAd. `Requirements` is an expression evaluated within the context of ads of `TargetType`. When `Requirements` evaluates to `True`, the matching ad is invalidated. A full example is given below.

Options

-help Display usage information

-version Display version information

-debug Print debugging information as the command executes.

-multiple Send more than one ClassAd, where the boundary between ClassAds is one or more blank lines.

-pool ***centralmanagerhostname[:portname]*** Specify a pool by giving the central manager's host name and an optional port number. The default is the `COLLECTOR_HOST` specified in the configuration file.

-tcp Use TCP for communication. Without this option, UDP is used.

General Remarks

The job and machine ClassAds are regularly updated. Therefore, the result of *condor_advertise* is likely to be overwritten in a very short time. It is unlikely that either Condor users (those who submit jobs) or administrators will ever have a use for this command. If it is desired to update or set a ClassAd attribute, the *condor_config_val* command is the proper command to use.

For those administrators who do need *condor_advertise*, the following attributes may be included:

`DaemonStartTime` - The time the service being advertised started running. Measured in seconds since the Unix epoch.

`UpdateSequenceNumber` - An integer that begins at 0 and increments by one each time the same ClassAd is again advertised.

If both of the above are included, the *condor_collector* will automatically include the following attributes:

`UpdatesTotal` - The actual number of advertisements for this daemon that the *condor_collector* has seen.

`UpdatesLost` - The number of advertisements that for this daemon that the *condor_collector* expected to see, but did not.

`UpdatesSequenced` - The total of `UpdatesTotal` and `UpdatesLost`.

`UpdatesHistory` - See `COLLECTOR_DAEMON_HISTORY_SIZE` in section 3.3.16.

Examples

Assume that a machine called condor.example.com is turned off, yet its *condor_startd* ClassAd does not expire for another 20 minutes. To avoid this machine being matched, an administrator chooses to delete the machine's *condor_startd* ClassAd. Create a file (called `remove_file` in this example) with the three required attributes:

```
MyType = "Query"
TargetType = "Machine"
Requirements = Name == "condor.example.com"
```

This file is used with the command:

```
% condor_advertise INVALIDATE_STARTD_ADS remove_file
```

Exit Status

condor_advertise will exit with a status value of 0 (zero) upon success, and it will exit with the value 1 (one) upon failure. Success means that all ClassAds were successfully sent to all *condor_collector* daemons. When there are multiple ClassAds or multiple *condor_collector* daemons, it is possible that some but not all publications succeed; in this case, the exit status is 1, indicating failure.

Author

Condor Team, University of Wisconsin–Madison

Copyright

condor_check_userlogs

Check user log files for errors

Synopsis

condor_check_userlogs *UserLogFile1* [*UserLogFile2 . . .UserLogFileN*]

Description

condor_check_userlogs is a program for checking a user log or set of users logs for errors. Output includes an indication that no errors were found within a log file, or a list of errors such as an execute or terminate event without a corresponding submit event, or multiple terminated events for the same job.

condor_check_userlogs is especially useful for debugging *condor_dagman* problems. If *condor_dagman* reports an error it is often useful to run *condor_check_userlogs* on the relevant log files.

Exit Status

condor_check_userlogs will exit with a status value of 0 (zero) upon success, and it will exit with the value 1 (one) upon failure.

Author

Condor Team, University of Wisconsin–Madison

Copyright

condor_checkpoint

send a checkpoint command to jobs running on specified hosts

Synopsis

condor_checkpoint [**-help** | **-version**]

condor_checkpoint [**-debug**] [**-pool** *centralmanagerhostname[:portnumber]*] [**-name** *hostname* | *hostname* | **-addr** *"<a.b.c.d:port>"* | *"<a.b.c.d:port>"* | **-constraint** *expression* | **-all**]

Description

condor_checkpoint sends a checkpoint command to a set of machines within a single pool. This causes the startd daemon on each of the specified machines to take a checkpoint of any running job that is executing under the standard universe. The job is temporarily stopped, a checkpoint is taken, and then the job continues. If no machine is specified, then the command is sent to the machine that issued the *condor_checkpoint* command.

The command sent is a periodic checkpoint. The job will take a checkpoint, but then the job will immediately continue running after the checkpoint is completed. *condor_vacate*, on the other hand, will result in the job exiting (vacating) after it produces a checkpoint.

If the job being checkpointed is running under the standard universe, the job produces a checkpoint and then continues running on the same machine. If the job is running under another universe, or if there is currently no Condor job running on that host, then *condor_checkpoint* has no effect.

There is generally no need for the user or administrator to explicitly run *condor_checkpoint*. Taking checkpoints of running Condor jobs is handled automatically following the policies stated in the configuration files.

Options

-help Display usage information

-version Display version information

-debug Causes debugging information to be sent to `stderr`, based on the value of the configuration variable `TOOL_DEBUG`

-pool ***centralmanagerhostname[:portnumber]*** Specify a pool by giving the central manager's host name and an optional port number

-name ***hostname*** Send the command to a machine identified by *hostname*

hostname Send the command to a machine identified by *hostname*

-addr *"<**a.b.c.d:port**>"* Send the command to a machine's master located at *"<a.b.c.d:port>"*

*"<**a.b.c.d:port**>"* Send the command to a machine located at *"<a.b.c.d:port>"*

-constraint ***expression*** Apply this command only to machines matching the given ClassAd *expression*

-all Send the command to all machines in the pool

Exit Status

condor_checkpoint will exit with a status value of 0 (zero) upon success, and it will exit with the value 1 (one) upon failure.

Examples

To send a *condor_checkpoint* command to two named machines:

```
% condor_checkpoint  robin cardinal
```

To send the *condor_checkpoint* command to a machine within a pool of machines other than the local pool, use the **-pool** option. The argument is the name of the central manager for the pool. Note that one or more machines within the pool must be specified as the targets for the command. This command sends the command to a the single machine named **cae17** within the pool of machines that has **condor.cae.wisc.edu** as its central manager:

```
% condor_checkpoint -pool condor.cae.wisc.edu -name cae17
```

Author

Condor Team, University of Wisconsin–Madison

Copyright

condor_chirp

Access files or job ClassAd from an executing job

Synopsis

condor_chirp **<Chirp-Command>**

Description

condor_chirp is not a command-line tool. *condor_chirp* is invoked by a Condor job, while the job is executing. It accesses files or job ClassAd attributes on the submit machine. Files can be read, written or removed. Job attributes can be read, and most attributes can be updated.

When invoked by a Condor job, the command-line arguments describe the operation to be performed. Each of these arguments is described below within the section on Chirp Commands. Descriptions using the terms *local* and *remote* are given from the point of view of the executing job.

If the input file name for **put** or **write** is a dash, *condor_chirp* uses standard input as the source. If the output file name for **fetch** is a dash, *condor_chirp* writes to standard output instead of a local file.

Jobs that use *condor_chirp* must have the attribute `WantIOProxy` set to `True` in the job ClassAd. To do this, place

```
+WantIOProxy = true
```

in the submit description file of the job.

condor_chirp only works for jobs run in the vanilla, parallel and java universes.

Chirp Commands

fetch ***RemoteFileName LocalFileName*** Copy the *RemoteFileName* from the submit machine to the execute machine, naming it *LocalFileName*.

put [-mode ***mode*****] [-perm** ***UnixPerm*****]** ***LocalFileName RemoteFileName*** Copy the *LocalFileName* from the execute machine to the submit machine, naming it *RemoteFileName*. The optional **-perm** *UnixPerm* argument describes the file access permissions in a Unix format; 660 is an example Unix format.

The optional **-mode** *mode* argument is one or more of the following characters describing the *RemoteFileName* file: `w`, open for writing; `a`, force all writes to append; `t`, truncate before

use; `c`, create the file, if it does not exist; `x`, fail if `c` is given and the file already exists.

remove *RemoteFileName* Remove the *RemoteFileName* file from the submit machine.

get_job_attr *JobAttributeName* Prints the named job ClassAd attribute to standard output.

set_job_attr *JobAttributeName AttributeValue* Sets the named job ClassAd attribute with the given attribute value.

ulog *Message* Appends *Message* to the job's user log.

read [-offset *offset*] [-stride *length skip*] *RemoteFileName Length* Read *Length* bytes from *RemoteFileName.* Optionally, implement a stride by starting the read at *offset* and reading *length* bytes with a stride of *skip* bytes.

write [-offset *offset*] [-stride *length skip*] *RemoteFileName LocalFileName* Write the contents of *LocalFileName* to *RemoteFileName*. Optionally, start writing to the remote file at *offset* and write *length* bytes with a stride of *skip* bytes.

rmdir [-r] *RemotePath* Delete the directory specified by *RemotePath.* If the optional **-r** is specified, recursively delete the entire directory.

getdir [-l] *RemotePath* List the contents of the directory specified by *RemotePath.* If *-l* is specified, list all metadata as well.

whoami Get the user's current identity.

whoareyou *RemoteHost* Get the identity of *RemoteHost.*

link [-s] *OldRemotePath NewRemotePath* Create a hard link from *OldRemotePath* to *NewRemotePath.* If the optional *-s* is specified, create a symbolic link instead.

readlink *RemoteFileName* Read the contents of the file defined by the symbolic link *RemoteFileName.*

stat *RemotePath* Get metadata for *RemotePath.* Examines the target, if it is a symbolic link.

lstat *RemotePath* Get metadata for *RemotePath*. Examines the file, if it is a symbolic link.

statfs *RemotePath* Get file system metadata for *RemotePath*.

access *RemotePath Mode* Check access permissions for *RemotePath*. *Mode* is one or more of the characters `r`, `w`, `x`, or `f`, representing read, write, execute, and existence, respectively.

chmod *RemotePath UnixPerm* Change the permissions of *RemotePath* to *UnixPerm*. *UnixPerm* describes the file access permissions in a Unix format; 660 is an example Unix format.

chown *RemotePath UID GID* Change the ownership of *RemotePath* to *UID* and *GID*. Changes the target of *RemotePath*, if it is a symbolic link.

chown *RemotePath UID GID* Change the ownership of *RemotePath* to *UID* and *GID*. Changes the link, if *RemotePath* is a symbolic link.

truncate *RemoteFileName Length* Truncates *RemoteFileName* to *Length* bytes.

utime *RemotePath AccessTime ModifyTime* Change the access to *AccessTime* and modification time to *ModifyTime* of *RemotePath*.

Examples

To copy a file from the submit machine to the execute machine while the user job is running, run

```
condor_chirp fetch remotefile localfile
```

To print to standard output the value of the `Requirements` expression from within a running job, run

```
condor_chirp get_job_attr Requirements
```

Note that the remote (submit-side) directory path is relative to the submit directory, and the local (execute-side) directory is relative to the current directory of the running program.

To append the word "foo" to a file called `RemoteFile` on the submit machine, run

```
echo foo | condor_chirp put -mode wat - RemoteFile
```

To append the message "Hello World" to the user log, run

```
condor_chirp ulog "Hello World"
```

Exit Status

condor_chirp will exit with a status value of 0 (zero) upon success, and it will exit with the value 1 (one) upon failure.

Author

Condor Team, University of Wisconsin–Madison

Copyright

condor_cod

manage COD machines and jobs

Synopsis

condor_cod [**-help** | **-version**]

condor_cod *request* [**-pool** *centralmanagerhostname[:portnumber]* | **-name** *scheddname*]| [**-addr** *"<a.b.c.d:port>"*] [[**-help** | **-version**] | [**-debug** | **-timeout N** | **-classad file**]][**-requirements expr**] [**-lease N**]

condor_cod *release* **-id** *ClaimID* [[**-help** | **-version**] | [**-debug** | **-timeout N** | **-classad file**]][**-fast**]

condor_cod *activate* **-id** *ClaimID* [[**-help** | **-version**] | [**-debug** | **-timeout N** | **-classad file**]][**-keyword string** | **-jobad filename** | **-cluster N** | **-proc N** | **-requirements expr**]

condor_cod *deactivate* **-id** *ClaimID* [[**-help** | **-version**] | [**-debug** | **-timeout N** | **-classad file**]][**-fast**]

condor_cod *suspend* **-id** *ClaimID* [[**-help** | **-version**] | [**-debug** | **-timeout N** | **-classad file**]]

condor_cod *renew* **-id** *ClaimID* [[**-help** | **-version**] | [**-debug** | **-timeout N** | **-classad file**]]

condor_cod *resume* **-id** *ClaimID* [[**-help** | **-version**] | [**-debug** | **-timeout N** | **-classad file**]]

condor_cod *delegate_proxy* **-id** *ClaimID* [[**-help** | **-version**] | [**-debug** | **-timeout N** | **-classad file**]][**-x509proxy** *ProxyFile*]

Description

condor_cod issues commands that manage and use COD claims on machines, given proper authorization.

Instead of specifying an argument of *request*, *release*, *activate*, *deactivate*, *suspend*, *renew*, or *resume*, the user may invoke the *condor_cod* tool by appending an underscore followed by one of these arguments. As an example, the following two commands are equivalent:

```
condor_cod release -id "<128.105.121.21:49973>#1073352104#4"

condor_cod_release -id "<128.105.121.21:49973>#1073352104#4"
```

To make these extended-name commands work, hard link the extended name to the *condor_cod* executable. For example on a Unix machine:

```
ln condor_cod_request condor_cod
```

The *request* argument gives a claim ID, and the other commands (*release*, *activate*, *deactivate*, *suspend*, and *resume*) use the claim ID. The claim ID is given as the last line of output for a *request*, and the output appears of the form:

```
ID of new claim is: "<a.b.c.d:portnumber>#x#y"
```

An actual example of this line of output is

```
ID of new claim is: "<128.105.121.21:49973>#1073352104#4"
```

Also see section 4.3 for more a complete description of COD.

Options

-help Display usage information

-version Display version information

-pool ***centralmanagerhostname[:portnumber]*** Specify a pool by giving the central manager's host name and an optional port number

-name ***scheddname*** Send the command to a machine identified by *scheddname*

-addr *"<**a.b.c.d:port**>"* Send the command to a machine located at *"<a.b.c.d:port>"*

-lease ***N*** For the **request** of a new claim, automatically release the claim after *N* seconds.

request Create a new COD claim

release Relinquish a claim and kill any running job

activate Start a job on a given claim

deactivate Kill the current job, but keep the claim

suspend Suspend the job on a given claim

renew Renew the lease to the COD claim

resume Resume the job on a given claim

delegate_proxy Delegate an X509 proxy for the given claim

General Remarks

Examples

Exit Status

condor_cod will exit with a status value of 0 (zero) upon success, and it will exit with the value 1 (one) upon failure.

Author

Condor Team, University of Wisconsin–Madison

Copyright

condor_cold_start

install and start Condor on this machine

Synopsis

***condor_cold_start* -help**

condor_cold_start [**-basedir** *directory*] [**-force**] [**-setuponly** | **-runonly**] [**-arch** *architecture*] [**-site** *repository*] [**-localdir** *directory*] [**-runlocalconfig** *file*] [**-logarchive** *archive*] [**-spoolarchive** *archive*] [**-execarchive** *archive*] [**-filelock**] [**-pid**] [**-artifact** *filename*] [**-wget**] [**-globuslocation** *directory*] **-configfile** *file*

Description

condor_cold_start installs and starts Condor on this machine, setting up or using a predefined configuration. In addition, it has the functionality to determine the local architecture if one is not specified. Additionally, this program can install pre-made `log`, `execute`, and/or `spool` directories by specifying the archived versions.

Options

-arch *architecturestr* Use the given *architecturestr* to fetch the installation package. The string is in the format:

<condor_version>-<machine_arch>-<os_name>-<os_version>

(for example 6.6.7-i686-Linux-2.4). The portion of this string <condor_version> may be replaced with the string "latest" (for example, latest-i686-Linux-2.4) to substitute the most recent version of Condor.

-artifact *filename* Use *filename* for name of the artifact file used to determine whether the *condor_master* daemon is still alive.

-basedir *directory* The directory to install or find the Condor executables and libraries. When not specified, the current working directory is assumed.

-execarchive *archive* Create the Condor `execute` directory from the given *archive* file.

-filelock Specifies that this program should use a POSIX file lock midwife program to create an artifact of the birth of a *condor_master* daemon. A file lock undertaker can later be used to

determine whether the *condor_master* daemon has exited. This is the preferred option when the user wants to check the status of the *condor_master* daemon from another machine that shares a distributed file system that supports POSIX file locking, for example, AFS.

-force Overwrite previously installed files, if necessary.

-globuslocation *directory* The location of the globus installation on this machine. When not specified `/opt/globus` is the directory used. This option is only necessary when other options of the form **-*archive** are specified.

-help Display brief usage information and exit.

-localdir *directory* The directory where the Condor `log`, `spool`, and `execute` directories will be installed. Each running instance of Condor must have its own local directory.

-logarchive *archive* Create the Condor log directory from the given *archive* file.

-pid This program is to use a unique process id midwife program to create an artifact of the birth of a *condor_master* daemon. A unique pid undertaker can later be used to determine whether the *condor_master* daemon has exited. This is the default option and the preferred method to check the status of the *condor_master* daemon from the same machine it was started on.

-runlocalconfig *file* A special local configuration file bound into the Condor configuration at runtime. This file only affects the instance of Condor started by this command. No other Condor instance sharing the same global configuration file will be affected.

-runonly Run Condor from the specified installation directory without installing it. It is possible to run several instantiations of Condor from a single installation.

-setuponly Install Condor without running it.

-site *repository* The ftp, http, gsiftp, or mounted file system directory where the installation packages can be found (for example, `www.cs.example.edu/packages/coldstart`).

-spoolarchive *archive* Create the Condor spool directory from the given *archive* file.

-wget Use *wget* to fetch the `log`, `spool`, and `execute` directories, if other options of the form **-*archive** are specified. *wget* must be installed on the machine and in the user's path.

-configfile *file* A required option to specify the Condor configuration file to use for this installation. This file can be located on an http, ftp, or gsiftp site, or alternatively on a mounted file system.

Exit Status

condor_cold_start will exit with a status value of 0 (zero) upon success, and non-zero otherwise.

Examples

To start a Condor installation on the current machine, using `http://www.example.com/Condor/deployment` as the installation site:

```
% condor_cold_start \
  -configfile http://www.example.com/Condor/deployment/condor_config.mobile \
  -site http://www.example.com/Condor/deployment
```

Optionally if this instance of Condor requires a local configuration file `condor_config.local`:

```
% condor_cold_start \
  -configfile http://www.example.com/Condor/deployment/condor_config.mobile \
  -site http://www.example.com/Condor/deployment \
  -runlocalconfig condor_config.local
```

See Also

condor_cold_stop (on page 711), *filelock_midwife* (on page 887), *uniq_pid_midwife* (on page 897).

Author

Condor Team, University of Wisconsin–Madison

Copyright

condor_cold_stop

reliably shut down and uninstall a running Condor instance

Synopsis

***condor_cold_stop* -help**

condor_cold_stop [**-force**] [**-basedir** *directory*] [**-localdir** *directory*] [**-runlocalconfig** *file*] [**-cleaninstall**] [**-cleanlocal**] [**-stop**] [**-logarchive** *archive*] [**-spoolarchive** *archive*] [**-execarchive** *archive*] [**-filelock**] [**-pid**] [**-artifact** *file*] [**-nogurl**] [**-globuslocation** *directory*] **-configfile** *file*

Description

condor_cold_stop reliably shuts down and uninstall a running Condor instance. This program first uses *condor_local_stop* to reliably shut down the running Condor instance. It then uses *condor_cleanup_local* to create and store archives of the `log`, `spool`, and `execute` directories. Its last task is to uninstall the Condor binaries and libraries using *cleanup_release*.

Options

-artifact *file* Uses *file* as the artifact file to determine whether the *condor_master* daemon is still alive.

-basedir *directory* Directory where the Condor installation can be found. When not specified, the current working directory is assumed.

-cleaninstall Remove the Condor installation. If none of the options **-cleaninstall**, **-cleanlocal**, or **-stop** are specified, the program behaves as though all of them have been provided.

-cleanlocal The program will remove the `log`, `spool`, `exec` directories for this Condor instance. If none of the options **-cleaninstall**, **-cleanlocal**, or **-stop** are specified, the program behaves as though all of them have been provided.

-configfile *file* The same configuration file path given to *condor_cold_start*. This program assumes the file is in the installation directory or the current working directory.

-execarchive *archive* The program will create a tar'ed and gzip'ed archive of the `execute` directory and stores it as *archive*. The *archive* can be a file path or a grid-ftp url.

-filelock Determine whether the *condor_master* daemon has exited using a file lock undertaker. This option must match the corresponding option given to *condor_cold_start*.

-force Ignore the status of the *condor_schedd* daemon (whether it has jobs in the queue or not) when shutting down Condor.

-globuslocation *directory* The directory containing the Globus installation. This option is required if any of the options of the form **-*archive** are used, and Globus is not installed in `/opt/globus`.

-localdir *directory* Directory where the `log`, `spool`, and `execute` directories are stored for this running instance of Condor. Required if the **-cleanlocal** option is specified.

-logarchive *archive* The program will create a tar'ed and gzip'ed archive of the `log` directory and stores it as *archive*. The *archive* can be a file path or a grid-ftp url.

-nogurl Do not use *globus-url-copy* to store the archives. This implies that the archives can only be stored on mounted file systems.

-pid Determine whether the *condor_master* daemon has exited using a unique process id undertaker. This option must match the corresponding option given to *condor_cold_start*.

-runlocalconfig *file* Bind *file* into the configuration used by this instance of Condor. This option should the one provided to *condor_cold_start*.

-spoolarchive *archive* The program will create a tar'ed and gzip'ed archive of the `spool` directory and stores it as *archive*. The *archive* can be a file path or a grid-ftp url.

-stop The program will shut down this running instance of Condor. If none of the options **-cleaninstall**, **-cleanlocal**, or **-stop** are specified, the program behaves as though all of them have been provided.

Exit Status

condor_cold_stop will exit with a status value of 0 (zero) upon success, and non-zero otherwise.

Examples

To shut down a Condor instance on the target machine:

```
% condor_cold_stop -configfile condor_config.mobile
```

To shutdown a Condor instance and archive the log directory:

```
% condor_cold_stop -configfile condor_config.mobile \
  -logarchive /tmp/log.tar.gz
```

See Also

condor_cold_start (on page 708), *filelock_undertaker* (on page 889), *uniq_pid_undertaker* (on page 899).

Author

Condor Team, University of Wisconsin–Madison

Copyright

condor_compile

create a relinked executable for submission to the Standard Universe

Synopsis

condor_compile *cc* | *CC* | *gcc* | *f77* | *g++* | *ld* | *make* | . . .

Description

Use *condor_compile* to relink a program with the Condor libraries for submission into Condor's Standard Universe. The Condor libraries provide the program with additional support, such as the capability to checkpoint, which is required in Condor's Standard Universe mode of operation. *condor_compile* requires access to the source or object code of the program to be submitted; if source or object code for the program is not available (i.e. only an executable binary, or if it is a shell script), then the program must submitted into Condor's Vanilla Universe. See the reference page for *condor_submit* and/or consult the "Condor Users and Administrators Manual" for further information.

To use *condor_compile*, simply enter "condor_compile" followed by whatever you would normally enter to compile or link your application. Any resulting executables will have the Condor libraries linked in. For example:

```
condor_compile cc -O -o myprogram.condor file1.c file2.c ...
```

will produce a binary "myprogram.condor" which is relinked for Condor, capable of checkpoint/migration/remote-system-calls, and ready to submit to the Standard Universe.

If the Condor administrator has opted to fully install *condor_compile*, then *condor_compile* can be followed by practically any command or program, including make or shell-script programs. For example, the following would all work:

```
condor_compile make

condor_compile make install

condor_compile f77 -O mysolver.f

condor_compile /bin/csh compile-me-shellscript
```

If the Condor administrator has opted to only do a partial install of *condor_compile*, the you are restricted to following *condor_compile* with one of these programs:

```
cc (the system C compiler)

c89 (POSIX compliant C compiler, on some systems)

CC (the system C++ compiler)

f77 (the system FORTRAN compiler)

gcc (the GNU C compiler)

g++ (the GNU C++ compiler)

g77 (the GNU FORTRAN compiler)

ld (the system linker)
```

NOTE: If you use explicitly call "ld" when you normally create your binary, simply use:

```
condor_compile ld <ld arguments and options>
```

instead.

Exit Status

condor_compile is a script that executes specified compilers and/or linkers. If an error is encountered before calling these other programs, *condor_compile* will exit with a status value of 1 (one). Otherwise, the exit status will be that given by the executed program.

Author

Condor Team, University of Wisconsin–Madison

Copyright

condor_config_bind

bind together a set of configuration files

Synopsis

***condor_config_bind* -help**

***condor_config_bind* -o** *outputfile configfile1 configfile2* [*configfile3...*]

Description

condor_config_bind dynamically binds two or more Condor configuration files through the use of a new configuration file. The purpose of this tool is to allow the user to dynamically bind a local configuration file into an already created, and possible immutable, configuration file. This is particularly useful when the user wants to modify a configuration but cannot actually make any changes to the global configuration file (even to change the list of local configuration files). This program does not modify the given configuration files. Rather, it creates a new configuration file that specifies the given configuration files as local configuration files.

Condor evaluates each of the configuration files in the given command-line order (left to right). A value defined in two or more of the configuration files results in the last one evaluated defining the value. It overrides any others. To bind a new local configuration into a global configuration, specify the local configuration second within the command-line ordering.

Options

configfile1 First configuration file to bind.

configfile2 Second configuration file to bind.

configfile3... An optional list of other configuration files to bind.

-help Display brief usage information and exit

-o *output_file* Specifies the file name where this program should output the binding configuration.

Exit Status

condor_config_bind will exit with a status value of 0 (zero) upon success, and non-zero on error.

Author

Condor Team, University of Wisconsin–Madison

Copyright

condor_config_val

Query or set a given Condor configuration variable

Synopsis

condor_config_val [**options**] [**-config**] [**-verbose**] *variable* [*variable . . .*]

condor_config_val [**options**] **-set** *string* [*string . . .*]

condor_config_val [**options**] **-rset** *string* [*string . . .*]

condor_config_val [**options**] **-unset** *variable* [*variable . . .*]

condor_config_val [**options**] **-runset** *variable* [*variable . . .*]

condor_config_val [**options**] **-tilde**

condor_config_val [**options**] **-owner**

condor_config_val [**options**] **-config**

condor_config_val [**options**] **-dump**

Description

condor_config_val can be used to quickly see what the current Condor configuration is on any given machine. Given a list of variables, *condor_config_val* will report what each of these variables is currently set to. If a given variable is not defined, *condor_config_val* will halt on that variable, and report that it is not defined. By default, *condor_config_val* looks in the local machine's configuration files in order to evaluate the variables.

condor_config_val can also be used to quickly set configuration variables for a specific daemon on a given machine. Each daemon remembers settings made by *condor_config_val*. The configuration file is not modified by this command. Persistent settings remain when the daemon is restarted. Runtime settings are lost when the daemon is restarted. In general, modifying a host's configuration with *condor_config_val* requires the `CONFIG` access level, which is disabled on all hosts by default. Administrators have more fine-grained control over which access levels can modify which settings. See section 3.6.1 on page 315 for more details on security settings.

The **-verbose** option displays the configuration file name and line number where a configuration variable is defined.

Any changes made by *condor_config_val* will not take effect until *condor_reconfig* is invoked.

It is generally wise to test a new configuration on a single machine to ensure that no syntax or other errors in the configuration have been made before the reconfiguration of many machines. Having

bad syntax or invalid configuration settings is a fatal error for Condor daemons, and they will exit. It is far better to discover such a problem on a single machine than to cause all the Condor daemons in the pool to exit.

The **-set** option sets one or more persistent configuration file entries. The *string* must be a single argument, so enclose it in double quote marks. A string must be of the form `"variable = value"`. Use of the **-set** option implies the use of configuration variables `SETTABLE_ATTRS...` (see 3.3.5), `ENABLE_PERSISTENT_CONFIG` (see 3.3.5), and `HOSTALLOW...` (see 3.3.5).

The **-rset** option sets one or more runtime configuration file entries. The *string* must be a single argument, so enclose it in double quote marks. A string must be of the form `"variable = value"`. Use of the **-rset** option implies the use of configuration variables `SETTABLE_ATTRS...` (see 3.3.5), `ENABLE_RUNTIME_CONFIG` (see 3.3.5), and `HOSTALLOW...` (see 3.3.5).

The **-unset** option changes one or more persistent configuration file entries to their previous value.

The **-runset** option changes one or more runtime configuration file entries to their previous value.

The **-tilde** option displays the path to the Condor home directory.

The **-owner** option displays the owner of the *condor_config_val* process.

The **-config** option displays the current configuration files in use.

The **-dump** option displays a list of all of the defined macros in the configuration files found by *condor_config_val*, along with their values. If the **-verbose** option is supplied as well, then the specific configuration file which defined each macro, along with the line number of its definition is also printed. NOTE: The output of this argument is likely to change in a future revision of Condor.

Options

-name *machine_name* Query the specified machine's *condor_master* daemon for its configuration. Does not function together with any of the options: **-dump**, **-config**, or **-verbose**.

-pool *centralmanagerhostname[:portnumber]* Use the given central manager and an optional port number to find daemons.

-address <***ip:port***> Connect to the given IP address and port number.

-master | **-schedd** | **-startd** | **-collector** | **-negotiator** The specific daemon to query.

-local-name Inspect the values of attributes that use local names.

Exit Status

condor_config_val will exit with a status value of 0 (zero) upon success, and it will exit with the value 1 (one) upon failure.

Examples

Here is a set of examples to show a sequence of operations using *condor_config_val*. To request the *condor_schedd* daemon on host perdita to display the value of the MAX_JOBS_RUNNING configuration variable:

```
% condor_config_val -name perdita -schedd MAX_JOBS_RUNNING
500
```

To request the *condor_schedd* daemon on host perdita to set the value of the MAX_JOBS_RUNNING configuration variable to the value 10.

```
% condor_config_val -name perdita -schedd -set "MAX_JOBS_RUNNING = 10"
Successfully set configuration "MAX_JOBS_RUNNING = 10" on
schedd perdita.cs.wisc.edu <128.105.73.32:52067>.
```

A command that will implement the change just set in the previous example.

```
% condor_reconfig -schedd perdita
Sent "Reconfig" command to schedd perdita.cs.wisc.edu
```

A re-check of the configuration variable reflects the change implemented:

```
% condor_config_val -name perdita -schedd MAX_JOBS_RUNNING
10
```

To set the configuration variable MAX_JOBS_RUNNING back to what it was before the command to set it to 10:

```
% condor_config_val -name perdita -schedd -unset MAX_JOBS_RUNNING
Successfully unset configuration "MAX_JOBS_RUNNING" on
schedd perdita.cs.wisc.edu <128.105.73.32:52067>.
```

A command that will implement the change just set in the previous example.

```
% condor_reconfig -schedd perdita
Sent "Reconfig" command to schedd perdita.cs.wisc.edu
```

A re-check of the configuration variable reflects that variable has gone back to is value before initial set of the variable:

```
% condor_config_val -name perdita -schedd MAX_JOBS_RUNNING
500
```

Author

Condor Team, University of Wisconsin–Madison

Copyright

condor_configure

Configure or install Condor

Synopsis

condor_configure* or *condor_install [**--help**]

condor_configure or ***condor_install*** [**--install[=<path/to/release>]**] [**--install-dir=<path>**] [**--prefix=<path>**] [**--local-dir=<path>**] [**--make-personal-condor**] [**--type = < submit, execute, manager >**] [**--central-manager = < hostname>**] [**--owner = < ownername >**] [**--make-personal-stork**] [**--overwrite**] [**--ignore-missing-libs**] [**--force**] [**--no-env-scripts**] [**--env-scripts-dir = < directory >**] [**--backup**] [**--stork**] [**--credd**] [**--verbose**]

Description

condor_configure and *condor_install* refer to a single script that installs and/or configures Condor on Unix machines. As the names imply, *condor_install* is intended to perform a Condor installation, and *condor_configure* is intended to configure (or reconfigure) an existing installation. Both will run with Perl 5.6.0 or more recent versions.

condor_configure (and *condor_install*) are designed to be run more than one time where required. It can install Condor when invoked with a correct configuration via

```
condor_install
```

or

```
condor_configure --install
```

or, it can change the configuration files when invoked via

```
condor_configure
```

Note that changes in the configuration files do not result in changes while Condor is running. To effect changes while Condor is running, it is necessary to further use the *condor_reconfig* or *condor_restart* command. *condor_reconfig* is required where the currently executing daemons need to be informed of configuration changes. *condor_restart* is required where the options **--make-personal-condor** or **--type** are used, since these affect which daemons are running.

Running *condor_configure* or *condor_install* with no options results in a usage screen being printed. The **--help** option can be used to display a full help screen.

Within the options given below, the phrase *release directories* is the list of directories that are released with Condor. This list includes: `bin`, `etc`, `examples`, `include`, `lib`, `libexec`, `man`, `sbin`, `sql` and `src`.

Options

—help Print help screen and exit

—install Perform installation, assuming that the current working directory contains the release directories. Without further options, the configuration is that of a Personal Condor, a complete one-machine pool. If used as an upgrade within an existing installation directory, existing configuration files and local directory are preserved. This is the default behavior of *condor_install*.

—install-dir=<**path**> Specifies the path where Condor should be installed or the path where it already is installed. The default is the current working directory.

—prefix=<**path**> This is an alias for **–install-dir**.

—local-dir=<**path**> Specifies the location of the local directory, which is the directory that generally contains the local (machine-specific) configuration file as well as the directories where Condor daemons write their run-time information (`spool`, `log`, `execute`). This location is indicated by the `LOCAL_DIR` variable in the configuration file. When installing (that is, if **–install** is specified), *condor_configure* will properly create the local directory in the location specified. If none is specified, the default value is given by the evaluation of `$(RELEASE_DIR)/local.$(HOSTNAME)`.

During subsequent invocations of *condor_configure* (that is, without the —install option), if the —local-dir option is specified, the new directory will be created and the `log`, `spool` and `execute` directories will be moved there from their current location.

—make-personal-condor Installs and configures for Personal Condor, a fully-functional, one-machine pool.

—type= < **submit, execute, manager** > One or more of the types may be listed. This determines the roles that a machine may play in a pool. In general, any machine can be a submit and/or execute machine, and there is one central manager per pool. In the case of a Personal Condor, the machine fulfills all three of these roles.

—central-manager=<**hostname**> Instructs the current Condor installation to use the specified machine as the central manager. This modifies the configuration variables

COLLECTOR_HOST and NEGOTIATOR_HOST to point to the given host name). The central manager machine's Condor configuration needs to be independently configured to act as a manager using the option **–type=manager**.

—owner=<**ownername**> Set configuration such that Condor daemons will be executed as the given owner. This modifies the ownership on the log, spool and execute directories and sets the CONDOR_IDS value in the configuration file, to ensure that Condor daemons start up as the specified effective user. See section 3.6.13 on UIDs in Condor on page 349 for details. This is only applicable when *condor_configure* is run by root. If not run as root, the owner is the user running the *condor_configure* command.

–overwrite Always overwrite the contents of the sbin directory in the installation directory. By default, *condor_install* will not install if it finds an existing sbin directory with Condor programs in it. In this case, *condor_install* will exit with an error message. Specify **–overwrite** or **–backup** to tell *condor_install* what to do.

This prevents *condor_install* from moving an sbin directory out of the way that it should not move. This is particularly useful when trying to install Condor in a location used by other things (/usr, /usr/local, etc.) For example: *condor_install* **–prefix=/usr** will not move /usr/sbin out of the way unless you specify the **–backup** option.

The **–backup** behavior is used to prevent *condor_install* from overwriting running daemons – Unix semantics will keep the existing binaries running, even if they have been moved to a new directory.

—backup Always backup the sbin directory in the installation directory. By default, *condor_install* will not install if it finds an existing sbin directory with Condor programs in it. In this case, *condor_install* with exit with an error message. You must specify **–overwrite** or **–backup** to tell *condor_install* what to do.

This prevents *condor_install* from moving an sbin directory out of the way that it should not move. This is particularly useful if you're trying to install Condor in a location used by other things (/usr, /usr/local, etc.) For example: *condor_install* **–prefix=/usr** will not move /usr/sbin out of the way unless you specify the **–backup** option.

The **–backup** behavior is used to prevent *condor_install* from overwriting running daemons – Unix semantics will keep the existing binaries running, even if they have been moved to a new directory.

—ignore-missing-libs Ignore missing shared libraries that are detected by *condor_install*. By default, *condor_install* will detect missing shared libraries such as libstdc++.so.5 on Linux; it will print messages and exit if missing libraries are detected. The **—ignore-missing-libs** will cause *condor_install* to not exit, and to proceed with the installation if missing libraries are detected.

—force This is equivalent to enabling both the **—overwrite** and **—ignore-missing-libs** command line options.

—no-env-scripts By default, *condor_configure* writes simple sh and csh shell scripts which can be sourced by their respective shells to set the user's `PATH` and `CONDOR_CONFIG` environment variables. This option prevents *condor_configure* from generating these scripts.

—env-scripts-dir=<**directory**> By default, the simple *sh* and *csh* shell scripts (see **—no-env-scripts** for details) are created in the root directory of the Condor installation. This option causes *condor_configure* to generate these scripts in the specified directory.

—make-personal-stork Creates a Personal Stork, using the *condor_credd* daemon.

—stork Configures the Stork data placement server. Use this option with the **—credd** option.

—credd Configure the the *condor_credd* daemon (credential manager daemon).

—verbose Print information about changes to configuration variables as they occur.

Exit Status

condor_configure will exit with a status value of 0 (zero) upon success, and it will exit with a nonzero value upon failure.

Examples

Install Condor on the machine (machine1@cs.wisc.edu) to be the pool's central manager. On machine1, within the directory that contains the unzipped Condor distribution directories:

```
% condor_install --type=submit,execute,manager
```

This will allow the machine to submit and execute Condor jobs, in addition to being the central manager of the pool.

To change the configuration such that machine2@cs.wisc.edu is an execute-only machine (that is, a dedicated computing node) within a pool with central manager on machine1@cs.wisc.edu, issue the command on that machine2@cs.wisc.edu from within the directory where Condor is installed:

```
% condor_configure --central-manager=machine1@cs.wisc.edu --type=execute
```

To change the location of the `LOCAL_DIR` directory in the configuration file, do (from the directory where Condor is installed):

```
% condor_configure --local-dir=/path/to/new/local/directory
```

This will move the `log`,`spool`,`execute` directories to `/path/to/new/local/directory` from the current local directory.

Author

Condor Team, University of Wisconsin–Madison

Copyright

condor_convert_history

Convert the history file to the new format

Synopsis

condor_convert_history [**-help**]

condor_convert_history *history-file1* [*history-file2...*]

Description

As of Condor version 6.7.19, the Condor history file has a new format to allow fast searches backwards through the file. Not all queries can take advantage of the speed increase, but the ones that can are significantly faster.

Entries placed in the history file after upgrade to Condor 6.7.19 will automatically be saved in the new format. The new format adds information to the string which distinguishes and separates job entries. In order to search within this new format, no changes are necessary. However, to be able to search the entire history, the history file must be converted to the updated format. *condor_convert_history* does this.

Turn the *condor_schedd* daemon off while converting history files. Turn it back on after conversion is completed.

Arguments to *condor_convert_history* are the history files to convert. The history file is normally in the Condor spool directory; it is named `history`. Since the history file is rotated, there may be multiple history files, and all of them should be converted. On Unix platform variants, the easiest way to do this is:

```
cd `condor_config_val SPOOL`
condor_convert_history history*
```

condor_convert_history makes a back up of each original history files in case of a problem. The names of these back up files are listed; names are formed by appending the suffix `.oldver` to the original file name. Move these back up files to a directory other than the spool directory. If kept in the spool directory, *condor_history* will find the back ups, and will appear to have duplicate jobs.

Exit Status

condor_convert_history will exit with a status value of 0 (zero) upon success, and it will exit with the value 1 (one) upon failure.

Author

Condor Team, University of Wisconsin–Madison

Copyright

condor_dagman

meta scheduler of the jobs submitted as the nodes of a DAG or DAGs

Synopsis

condor_dagman [**-debug** *level*] [**-rescue** *filename*] [**-maxidle** *numberOfJobs*] [**-maxjobs** *numberOfJobs*] [**-maxpre** *NumberOfPREscripts*] [**-maxpost** *NumberOfPOSTscripts*] [**-noeventchecks**] [**-allowlogerror**] [**-usedagdir**] **-lockfile** *filename* [**-waitfordebug**] [**-autorescue** *0|1*] [**-dorescuefrom** *number*] **-csdversion** *version_string* [**-allowversionmismatch**] [**-DumpRescue**] [**-verbose**] [**-force**] [**-notification** *value*] [**-dagman** *DagmanExecutable*] [**-outfile_dir** *directory*] [**-update_submit**] [**-import_env**] **-dag** *dag_file* [***-dag*** *dag_file_2 . . .**-dag*** *dag_file_n*]

Description

condor_dagman is a meta scheduler for the Condor jobs within a DAG (directed acyclic graph) (or multiple DAGs). In typical usage, a submitter of jobs that are organized into a DAG submits the DAG using *condor_submit_dag*. *condor_submit_dag* does error checking on aspects of the DAG and then submits *condor_dagman* as a Condor job. *condor_dagman* uses log files to coordinate the further submission of the jobs within the DAG.

As part of *daemoncore*, the set of command-line arguments given in section 3.9.2 work for *condor_dagman*.

Arguments to *condor_dagman* are either automatically set by *condor_submit_dag* or they are specified as command-line arguments to *condor_submit_dag* and passed on to *condor_dagman*. The method by which the arguments are set is given in their description below.

condor_dagman can run multiple, independent DAGs. This is done by specifying multiple **-dag** *a*rguments. Pass multiple DAG input files as command-line arguments to *condor_submit_dag*.

Debugging output may be obtained by using the **-debug** *level* option. Level values and what they produce is described as

- level = 0; never produce output, except for usage info
- level = 1; very quiet, output severe errors
- level = 2; normal output, errors and warnings
- level = 3; output errors, as well as all warnings
- level = 4; internal debugging output
- level = 5; internal debugging output; outer loop debugging

- level = 6; internal debugging output; inner loop debugging
- level = 7; internal debugging output; rarely used

Options

-debug *level* An integer level of debugging output. *level* is an integer, with values of 0-7 inclusive, where 7 is the most verbose output. This command-line option to *condor_submit_dag* is passed to *condor_dagman* or defaults to the value 3, as set by *condor_submit_dag*.

-rescue *filename* Sets the file name of the rescue DAG to write in the case of a failure. As passed by *condor_submit_dag*, the name of the file will be the name of the DAG input file concatenated with the string `.rescue`. This argument is now optional, and in general it is preferred to not specify it. This allows *condor_dagman* to automatically generate an appropriate rescue DAG name.

-maxidle *NumberOfJobs* Sets the maximum number of idle jobs allowed before *condor_dagman* stops submitting more jobs. If DAG nodes have a cluster with more than one job in it, each job in the cluster is counted individually. Once idle jobs start to run, *condor_dagman* will resume submitting jobs. *NumberOfJobs* is a positive integer. This command-line option to *condor_submit_dag* is passed to *condor_dagman*. If not specified, the number of idle jobs is unlimited.

-maxjobs *numberOfJobs* Sets the maximum number of clusters within the DAG that will be submitted to Condor at one time. *numberOfJobs* is a positive integer. This command-line option to *condor_submit_dag* is passed to *condor_dagman*. If not specified, the default number of clusters is unlimited. If a cluster contains more than one job, only the cluster is counted for purposes of **maxjobs**.

-maxpre *NumberOfPREscripts* Sets the maximum number of PRE scripts within the DAG that may be running at one time. *NumberOfPREScripts* is a positive integer. This command-line option to *condor_submit_dag* is passed to *condor_dagman*. If not specified, the default number of PRE scripts is unlimited.

-maxpost *NumberOfPOSTscripts* Sets the maximum number of POST scripts within the DAG that may be running at one time. *NumberOfPOSTScripts* is a positive integer. This command-line option to *condor_submit_dag* is passed to *condor_dagman*. If not specified, the default number of POST scripts is unlimited.

-noeventchecks This argument is no longer used; it is now ignored. Its functionality is now implemented by the `DAGMAN_ALLOW_EVENTS` configuration macro (see section 3.3.26).

-allowlogerror This optional argument has *condor_dagman* try to run the specified DAG, even in the case of detected errors in the user log specification. As of version 7.3.2, this argument has an effect only on DAGs containing Stork job nodes.

-usedagdir This optional argument causes *condor_dagman* to run each specified DAG as if the directory containing that DAG file was the current working directory. This option is most useful when running multiple DAGs in a single *condor_dagman*.

-lockfile *filename* Names the file created and used as a lock file. The lock file prevents execution of two of the same DAG, as defined by a DAG input file. A default lock file ending with the suffix `.dag.lock` is passed to *condor_dagman* by *condor_submit_dag*.

-waitfordebug This optional argument causes *condor_dagman* to wait at startup until someone attaches to the process with a debugger and sets the wait_for_debug variable in main_init() to false.

-autorescue *0|1* Whether to automatically run the newest rescue DAG for the given DAG file, if one exists (0 = `false`, 1 = `true`).

-dorescuefrom *number* Forces *condor_dagman* to run the specified rescue DAG number for the given DAG. A value of 0 is the same as not specifying this option. Specifying a nonexistent rescue DAG is a fatal error.

-csdversion *version_string* *version_string* is the version of the *condor_submit_dag* program. At startup, *condor_dagman* checks for a version mismatch with the *condor_submit_dag* version in this argument.

-allowversionmismatch This optional argument causes *condor_dagman* to allow a version mismatch between *condor_dagman* itself and the `.condor.sub` file produced by *condor_submit_dag* (or, in other words, between *condor_submit_dag* and *condor_dagman*). WARNING! This option should be used only if absolutely necessary. Allowing version mismatches can cause subtle problems when running DAGs. (Note that, starting with version 7.4.0, *condor_dagman* no longer requires an exact version match between itself and the `.condor.sub` file. Instead, a "minimum compatible version" is defined, and any `.condor.sub` file of that version or newer is accepted.)

-DumpRescue This optional argument causes *condor_dagman* to immediately dump a Rescue DAG and then exit, as opposed to actually running the DAG. This feature is mainly intended for testing. The Rescue DAG file is produced whether or not there are parse errors reading the original DAG input file. The name of the file differs if there was a parse error.

-verbose (This argument is included only to be passed to *condor_submit_dag* if lazy submit file generation is used for nested DAGs.) Cause *condor_submit_dag* to give verbose error messages.

-force (This argument is included only to be passed to *condor_submit_dag* if lazy submit file generation is used for nested DAGs.) Require *condor_submit_dag* to overwrite the files that it produces, if the files already exist. Note that `dagman.out` will be appended to, not overwritten. If new-style rescue DAG mode is in effect, and any new-style rescue DAGs exist, the **-force** flag will cause them to be renamed, and the original DAG will be run. If old-style rescue DAG mode is in effect, any existing old-style rescue DAGs will be deleted, and the original DAG will be run. Section 2.10.7 details rescue DAGs.

-notification *value* (This argument is only included to be passed to *condor_submit_dag* if lazy submit file generation is used for nested DAGs.) Sets the e-mail notification for DAGMan itself. This information will be used within the Condor submit description file for DAGMan. This file is produced by *condor_submit_dag*. See **notification** within the section of submit description file commands in the *condor_submit* manual page on page 825 for specification of *value*.

-dagman *DagmanExecutable* (This argument is included only to be passed to *condor_submit_dag* if lazy submit file generation is used for nested DAGs.) Allows the specification of an alternate *condor_dagman* executable to be used instead of the one found in the user's path. This must be a fully qualified path.

-outfile_dir *directory* (This argument is included only to be passed to *condor_submit_dag* if lazy submit file generation is used for nested DAGs.) Specifies the directory in which the `.dagman.out` file will be written. The *directory* may be specified relative to the current working directory as *condor_submit_dag* is executed, or specified with an absolute path. Without this option, the `.dagman.out` file is placed in the same directory as the first DAG input file listed on the command line.

-update_submit (This argument is included only to be passed to *condor_submit_dag* if lazy submit file generation is used for nested DAGs.) This optional argument causes an existing `.condor.sub` file to not be treated as an error; rather, the `.condor.sub` file will be overwritten, but the existing values of **-maxjobs**, **-maxidle**, **-maxpre**, and **-maxpost** will be preserved.

-import_env (This argument is included only to be passed to *condor_submit_dag* if lazy submit file generation is used for nested DAGs.) This optional argument causes *condor_submit_dag* to import the current environment into the **environment** command of the `.condor.sub` file it generates.

-dag *filename* *filename* is the name of the DAG input file that is set as an argument to *condor_submit_dag*, and passed to *condor_dagman*.

Exit Status

condor_dagman will exit with a status value of 0 (zero) upon success, and it will exit with the value 1 (one) upon failure.

Examples

condor_dagman is normally not run directly, but submitted as a Condor job by running condor_submit_dag. See the condor_submit_dag manual page 859 for examples.

Author

Condor Team, University of Wisconsin–Madison

Copyright

condor_fetchlog

Retrieve a daemon's log file that is located on another computer

Synopsis

condor_fetchlog [**-help** | **-version**]

condor_fetchlog [**-pool** *centralmanagerhostname[:portnumber]*] [**-master** | **-startd** | **-schedd** | **-collector** | **-negotiator** | **-kbdd**] *machine-name subsystem[.extension]*

Description

condor_fetchlog contacts Condor running on the machine specified by *machine-name,* and asks it to return a log file from that machine. Which log file is determined from the *subsystem[.extension]* argument. The log file is printed to standard output. This command eliminates the need to remotely log in to a machine in order to retrieve a daemon's log file.

For security purposes of authentication and authorization, this command requires an administrator's level of access. See section 3.6.1 on page 315 for more details about Condor's security mechanisms.

The *subsystem[.extension]* argument is utilized to construct the log file's name. Without an optional *.extension,* the value of the configuration variable named *subsystem*_LOG defines the log file's name. When specified, the *.extension* is appended to this value.

Typical strings for the argument *subsystem* are as given as possible values of the predefined configuration variable `$(SUBSYSTEM)`. See the definition in section 3.3.1. Note that access to any additional logs can be enabled by simply specifying the path to each log in the configuration file with a configuration parameter named `<NAME>_LOG`, choosing an arbitrary unique name for each case.

A value for the optional *.extension* argument is typically one of the three strings:

1. `.old`
2. `.slot<X>`
3. `.slot<X>.old`

Within these strings, `<X>` is substituted with the slot number.

Options

-help Display usage information

-version Display version information

-pool ***centralmanagerhostname[:portnumber]*** Specify a pool by giving the central manager's host name and an optional port number

-master Send the command to the *condor_master* daemon (default)

-startd Send the command to the *condor_startd* daemon

-schedd Send the command to the *condor_schedd* daemon

-collector Send the command to the *condor_collector* daemon

-kbdd Send the command to the *condor_kbdd* daemon

Examples

To get the *condor_negotiator* daemon's log from a host named `head.example.com` from within the current pool:

```
condor_fetchlog head.example.com NEGOTIATOR
```

To get the *condor_startd* daemon's log from a host named `execute.example.com` from within the current pool:

```
condor_fetchlog execute.example.com STARTD
```

This command requested the *condor_startd* daemon's log from the *condor_master*. If the *condor_master* has crashed or is unresponsive, ask another daemon running on that computer to return the log. For example, ask the *condor_startd* daemon to return the *condor_master*'s log:

```
condor_fetchlog -startd execute.example.com MASTER
```

Exit Status

condor_fetchlog will exit with a status value of 0 (zero) upon success, and it will exit with the value 1 (one) upon failure.

Author

Condor Team, University of Wisconsin–Madison

Copyright

condor_findhost

find machine(s) in the pool that can be used with minimal impact on currently running Condor jobs and best meet any specified constraints

Synopsis

condor_findhost [**-help**] [**-m**] [**-n** *num*] [**-c** *c_expr*] [**-r** *r_expr*] [**-p** *centralmanagerhostname*]

Description

condor_findhost searches a Condor pool of machines for the best machine or machines that will have the minimum impact on running Condor jobs if the machine or machines are taken out of the pool. The search may be limited to the machine or machines that match a set of constraints and rank expression.

condor_findhost returns a fully-qualified domain name for each machine. The search is limited (constrained) to a specific set of machines using the *-c* option. The search can use the *-r* option for rank, the criterion used for selecting a machine or machines from the constrained list.

Options

-help Display usage information and exit

-m Only search for entire machines. Slots within an entire machine are not considered.

-n *num* Find and list up to *num* machines that fulfill the specification. *num* is an integer greater than zero.

-c *c_expr* Constrain the search to only consider machines that result from the evaluation of *c_expr*. *c_expr* is a ClassAd expression.

-r *r_expr* *r_expr* is the rank expression evaluated to use as a basis for machine selection. *r_expr* is a ClassAd expression.

-p *centralmanagerhostname* Specify the pool to be searched by giving the central manager's host name. Without this option, the current pool is searched.

General Remarks

condor_findhost is used to locate a machine within a pool that can be taken out of the pool with the least disturbance of the pool.

An administrator should set preemption requirements for the Condor pool. The expression

```
(Interactive =?= TRUE )
```

will let *condor_findhost* know that it can claim a machine even if Condor would not normally preempt a job running on that machine.

Exit Status

The exit status of *condor_findhost* is zero on success. If not able to identify as many machines as requested, it returns one more than the number of machines identified. For example, if 8 machines are requested, and *condor_findhost* only locates 6, the exit status will be 7. If not able to locate any machines, or an error is encountered, *condor_findhost* will return the value 1.

Examples

To find and list four machines, preferring those with the highest mips (on Drystone benchmark) rating:

```
condor_findhost -n 4 -r "mips"
```

To find and list 24 machines, considering only those where the `kflops` attribute is not defined:

```
condor_findhost -n 24 -c "kflops=?=undefined"
```

Author

Condor Team, University of Wisconsin–Madison

Copyright

condor_glidein

add a remote grid resource to a local Condor pool

Synopsis

condor_glidein [**-help**]

condor_glidein [**-admin** *address*] [**-anybody**] [**-archdir** *dir*] [**-basedir** *basedir*] [**-count** *CPU count*] [**<Execute Task Options>**] [**<Generate File Options>**] [**-gsi_daemon_name** *cert_name*] [**-idletime** *minutes*] [**-install_gsi_trusted_ca_dir** *path*] [**-install_gsi_gridmap** *file*] [**-localdir** *dir*] [**-memory** *MBytes*] [**-project** *name*] [**-queue** *name*] [**-runtime** *minutes*] [**-runonly**] [**<Set Up Task Options>**] [**-suffix** *suffix*] [**-slots** *slot count*] *<contact argument>*

Description

condor_glidein allows the temporary addition of a grid resource to a local Condor pool. The addition is accomplished by installing and executing some of the Condor daemons on the remote grid resource, such that it reports in as part of the local Condor pool. *condor_glidein* accomplishes two separate tasks: set up and execution. These separated tasks allow flexibility, in that the user may use *condor_glidein* to do only one of the tasks or both, in addition to customizing the tasks.

The set up task generates a script that may be used to start the Condor daemons during the execution task, places this script on the remote grid resource, composes and installs a configuration file, and it installs the *condor_master*, *condor_startd* and *condor_starter* daemons on the grid resource.

The execution task runs the script generated by the set up task. The goal of the script is to invoke the *condor_master* daemon. The Condor job *glidein_startup* appears in the queue of the local Condor pool for each invocation of *condor_glidein*. To remove the grid resource from the local Condor pool, use *condor_rm* to remove the *glidein_startup* job.

The Condor jobs to do both the set up and execute tasks utilize Condor-G and Globus gt2 protocols to communicate with the remote resource. Therefore, an X.509 certificate (proxy) is required for the user running *condor_glidein*.

Specify the remote grid machine with the command line argument *<contact argument>*. *<contact argument>* takes one of 4 forms:

1. *hostname*
2. *Globus contact string*
3. *hostname/jobmanager-<schedulername>*

4. **-contactfile** *filename*

The argument **-contactfile** *filename* specifies the full path and file name of a file that contains Globus contact strings. Each of the resources given by a Globus contact string is added to the local Condor pool.

The set up task of *condor_glidein* copies the binaries for the correct platform from a central server. To obtain access to the server, or to set up your own server, follow instructions on the Glidein Server Setup page, at http://www.cs.wisc.edu/condor/glidein. Set up need only be done once per site, as the installation is never removed.

By default, all files installed on the remote grid resource are placed in the directory `$(HOME)/Condor_glidein`. `$(HOME)` is evaluated and defined on the remote machine using a grid map. This directory must be in a shared file system accessible by all machines that will run the Condor daemons. By default, the daemon's log files will also be written in this directory. Change this directory with the **-localdir** option to make Condor daemons write to local scratch space on the execution machine. For debugging initial problems, it may be convenient to have the log files in the more accessible default directory. If using the default directory, occasionally clean up old log and execute directories to avoid running out of space.

Examples

To have 10 grid resources running PBS at a grid site with a gatekeeper named gatekeeper.site.edu join the local Condor pool:

```
% condor_glidein -count 10 gatekeeper.site.edu/jobmanager-pbs
```

If you try something like the above and *condor_glidein* is not able to automatically determine everything it needs to know about the remote site, it will ask you to provide more information. A typical result of this process is something like the following command:

```
% condor_glidein \
    -count 10 \
    -arch 6.6.7-i686-pc-Linux-2.4 \
    -setup_jobmanager jobmanager-fork \
    gatekeeper.site.edu/jobmanager-pbs
```

The Condor jobs that do the set up and execute tasks will appear in the queue for the local Condor pool. As a result of a successful glidein, use *condor_status* to see that the remote grid resources are part of the local Condor pool.

A list of common problems and solutions is presented in this manual page.

Generate File Options

-genconfig Create a local copy of the configuration file that may be used on the remote resource. The file is named `glidein_condor_config.<suffix>`. The string defined by <suffix> defaults to the process id (PID) of the *condor_glidein* process or is defined with the **-suffix** command line option. The configuration file may be edited for later use with the **-useconfig** option.

-genstartup Create a local copy of the script used on the remote resource to invoke the *condor_master*. The file is named `glidein_startup.<suffix>`. The string defined by <suffix> defaults to the process id (PID) of the *condor_glidein* process or is defined with the **-suffix** command line option. The file may be edited for later use with the **-usestartup** option.

-gensubmit Generate submit description files, but do not submit. The submit description file for the set up task is named `glidein_setup.submit.<suffix>`. The submit description file for the execute task is named `glidein_run.submit.<suffix>`. The string defined by <suffix> defaults to the process id (PID) of the *condor_glidein* process or is defined with the **-suffix** command line option.

Set Up Task Options

-setuponly Do only the set up task of *condor_glidein*. This option cannot be run simultaneously with **-runonly**.

-setup_here Do the set up task on the local machine, instead of at a remote grid resource. This may be used, for example, to do the set up task of *condor_glidein* in an AFS area that is read-only from the remote grid resource.

-forcesetup During the set up task, force the copying of files, even if this overwrites existing files. Use this to push out changes to the configuration.

-useconfig *config_file* The set up task copies the specified configuration file, rather than generating one.

-usestartup *startup_file* The set up task copies the specified startup script, rather than generating one.

-setup_jobmanager *jobmanagername* Identifies the jobmanager on the remote grid resource to receive the files during the set up task. If a reasonable default can be discovered through

MDS, this is optional. *jobmanagername* is a string representing any gt2 name for the job manager. The correct string in most cases will be *jobmanager-fork*. Other common strings may be *jobmanager*, *jobmanager-condor*, *jobmanager-pbs*, and *jobmanager-lsf*.

Execute Task Options

-runonly Starts execution of the Condor daemons on the grid resource. If any of the necessary files or executables are missing, *condor_glidein* exits with an error code. This option cannot be run simultaneously with **-setuponly**.

-run_here Runs *condor_master* directly rather than submitting a Condor job that causes the remote execution. To instead generate a script that does this, use **-run_here** in combination with **-gensubmit**. This may be useful for running Condor daemons on resources that are not directly accessible by Condor.

Options

-help Display brief usage information and exit.

-basedir ***basedir*** Specifies the base directory on the remote grid resource used for placing files. The default directory is `$(HOME)/Condor_glidein` on the grid resource.

-archdir ***dir*** Specifies the directory on the remote grid resource for placement of the Condor executables. The default value for **-archdir** *i*s based upon version information on the grid resource. It is of the form <`basedir`>/<`condor-version`>-<`Globus canonicalsystemname`>. An example of the directory (without the base directory) for Condor version 7.6.0 running on a 64-bit Intel processor with RHEL 3 is `7.6.0-x86_64-pc-Linux-2.4-glibc2.3`.

-localdir ***dir*** Specifies the directory on the remote grid resource in which to create log and execution subdirectories needed by Condor. If limited disk quota in the home or base directory on the grid resource is a problem, set **-localdir** to a large temporary space, such as `/tmp` or `/scratch`. If the batch system requires invocation of Condor daemons in a temporary scratch directory, '.' may be used for the definition of the **-localdir** option.

-arch ***architecture*** Identifies the platform of the required tarball containing the correct Condor daemon executables to download and install. If a reasonable default can be discovered through MDS, this is optional. A list of possible values may be found at http://www.cs.wisc.edu/condor/glidein/binaries. The architecture name is the same as the

tarball name without the suffix `tar.gz`. An example is 6.6.5-i686-pc-Linux-2.4 .

-queue *name* The argument *name* is a string used at the grid resource to identify a job queue.

-project *name* The argument *name* is a string used at the grid resource to identify a project name.

-memory *MBytes* The maximum memory size in Megabytes to request from the grid resource.

-count *CPU count* The number of CPUs requested to join the local pool. The default is 1.

-slots *slot count* For machines with multiple CPUs, the CPUs maybe divided up into slots. *slot count* is the number of slots that results. By default, Condor divides multiple-CPU resources such that each CPU is a slot, each with an equal share of RAM, disk, and swap space. This option configures the number of slots, so that multi-threaded jobs can run in a slot with multiple CPUs. For example, if 4 CPUs are requested and **-slots** is not specified, Condor will divide the request up into 4 slots with 1 CPU each. However, if **-slots** *2* is specified, Condor will divide the request up into 2 slots with 2 CPUs each, and if **-slots** *1* is specified, Condor will put all 4 CPUs into one slot.

-idletime *minutes* The amount of time that a remote grid resource will remain idle state, before the daemons shut down. A value of 0 (zero) means that the daemons never shut down due to remaining in the idle state. In this case, the **-runtime** option defines when the daemons shut down. The default value is 20 minutes.

-runtime *minutes* The maximum amount of time the Condor daemons on the remote grid resource will run before shutting themselves down. This option is useful for resources with enforced maximum run times. Setting **-runtime** to be a few minutes shorter than the enforced limit gives the daemons time to perform a graceful shut down.

-*anybody* Sets the Condor `START` expression for the added remote grid resource to `True`. This permits any user's job which can run on the added remote grid resource to run. Without this option, only jobs owned by the user executing *condor_glidein* can execute on the remote grid resource. WARNING: Using this option may violate the usage policies of many institutions.

-admin *address* Where to send e-mail with problems. The default is the login of the user running *condor_glidein* at UID domain of the local Condor pool.

-suffix *X* Suffix to use when generating files. Default is process id.

-gsi_daemon_name *cert_name* Includes and enables GSI authentication in the configuration for the remote grid resource. The argument is the GSI certificate name that the daemons will use to authenticate themselves.

-install_gsi_trusted_ca_dir *path* The argument identifies the directory containing the trusted CA certificates that the daemons are to use (for example, `/etc/grid-security/certificates`). The contents of this directory will be installed at the remote site in the directory <`basedir`>`/grid-security`.

-install_gsi_gridmap *file* The argument is the file name of the GSI-specific X.509 map file that the daemons will use. The file will be installed at the remote site in <`basedir`>`/grid-security`. The file contains entries mapping certificates to user names. At the very least, it must contain an entry for the certificate given by the command-line option **-gsi_daemon_name** . If other Condor daemons use different certificates, then this file will also list any certificates that the daemons will encounter for the *condor_schedd*, *condor_collector*, and *condor_negotiator*. See section 3.6.3 for more information.

Exit Status

condor_glidein will exit with a status value of 0 (zero) upon complete success, or with non-zero values upon failure. The status value will be 1 (one) if *condor_glidein* encountered an error making a directory, was unable to copy a tar file, encountered an error in parsing the command line, or was not able to gather required information. The status value will be 2 (two) if there was an error in the remote set up. The status value will be 3 (three) if there was an error in remote submission. The status value will be -1 (negative one) if no resource was specified in the command line.

Common problems are listed below. Many of these are best discovered by looking in the `StartLog` log file on the remote grid resource.

WARNING: The file xxx is not writable by condor This error occurs when *condor_glidein* is run in a directory that does not have the proper permissions for Condor to access files. An AFS directory does not give Condor the user's AFS ACLs.

Glideins fail to run due to GLIBC errors Check the list of available glidein binaries (http://www.cs.wisc.edu/condor/glidein/binaries), and try specifying the architecture name that includes the correct glibc version for the remote grid site.

Glideins join pool but no jobs run on them One common cause of this problem is that the remote grid resources are in a different file system domain, and the submitted Condor jobs have an implicit requirement that they must run in the same file system domain. See section 2.5.4 for details on using Condor's file transfer capabilities to solve this problem. Another cause of this problem is a communication failure. For example, a firewall may be preventing the

condor_negotiator or the *condor_schedd* daemons from connecting to the *condor_startd* on the remote grid resource. Although work is being done to remove this requirement in the future, it is currently necessary to have full bidirectional connectivity, at least over a restricted range of ports. See page 182 for more information on configuring a port range.

Glideins run but fail to join the pool This may be caused by the local pool's security settings or by a communication failure. Check that the security settings in the local pool's configuration file allow write access to the remote grid resource. To not modify the security settings for the pool, run a separate pool specifically for the remote grid resources, and use flocking to balance jobs across the two pools of resources. If the log files indicate a communication failure, then see the next item.

The startd cannot connect to the collector This may be caused by several things. One is a firewall. Another is when the compute nodes do not have even outgoing network access. Configuration to work without full network access to and from the compute nodes is still in the experimental stages, so for now, the short answer is that you must at least have a range of open (bidirectional) ports and set up the configuration file as described on page 182. Use the option **-genconfig**, edit the generated configuration file, and then do the glidein execute task with the option **-useconfig**.)

Another possible cause of connectivity problems may be the use of UDP by the *condor_startd* to register itself with the *condor_collector*. Force it to use TCP as described on page 182.

Yet another possible cause of connectivity problems is when the remote grid resources have more than one network interface, and the default one chosen by Condor is not the correct one. One way to fix this is to modify the glidein startup script using the **-genstartup** and **-usestartup** options. The script needs to determine the IP address associated with the correct network interface, and assign this to the environment variable `_condor_NETWORK_INTERFACE`.

NFS file locking problems If the **-localdir** option uses files on NFS (not recommended, but sometimes convenient for testing), the Condor daemons may have trouble manipulating file locks. Try inserting the following into the configuration file:

```
IGNORE_NFS_LOCK_ERRORS = True
```

Author

Condor Team, University of Wisconsin–Madison

Copyright

condor_history

View log of Condor jobs completed to date

Synopsis

condor_history [**-help**]

condor_history [**-backwards**] [**-completedsince** *postgrestimestamp*] [**-constraint** *expr*] [**-f** *filename*] [**-format** *formatString AttributeName*] [**-l** | **-long** | **-xml**] [**-match** *number*] [**-name** *schedd-name*] [**cluster** | **cluster.process** | **owner**]

Description

condor_history displays a summary of all Condor jobs listed in the specified history files, or in the Quill database, when Quill is enabled. If no history files are specified (with the **-f** option) and Quill is not enabled, the local history file as specified in Condor's configuration file (`$(SPOOL)/history` by default) is read. The default listing summarizes (in chronological order) each job on a single line, and contains the following items:

ID The cluster/process id of the job.

OWNER The owner of the job.

SUBMITTED The month, day, hour, and minute the job was submitted to the queue.

RUN_TIME Remote wall clock time accumulated by the job to date in days, hours, minutes, and seconds. See the definition of `RemoteWallClockTime` on page 910.

ST Completion status of the job (C = completed and X = removed).

COMPLETED The time the job was completed.

CMD The name of the executable.

If a job ID (in the form of *cluster_id* or *cluster_id.proc_id*) or an *owner* is provided, output will be restricted to jobs with the specified IDs and/or submitted by the specified owner. The *-constraint* option can be used to display jobs that satisfy a specified boolean expression.

The history file is kept in chronological order, implying that new entries are appended at the end of the file. As of Condor version 6.7.19, the format of the history file is altered to enable faster reading of the history file backwards (most recent job first). History files written with earlier versions of Condor, as well as those that have entries of both the older and newer format need to be converted to the new format. See the *condor_convert_history* manual page on page 727 for details on converting history files to the new format.

Options

-help Display usage information and exit.

-backwards List jobs in reverse chronological order. The job most recently added to the history file is first.

-completedsince *postgrestimestamp* When Quill is enabled, display only job ads that were in the Completed job state on or after the date and time given by the *postgrestimestamp*. The *postgrestimestamp* follows the syntax as given for *PostgreSQL* version 8.0. The behavior of this option is undefined when Quill is *not* enabled.

-constraint *expr* Display jobs that satisfy the expression.

-f *filename* Use the specified file instead of the default history file. When Quill is enabled, this option will force the query to read from the history file, and not the database.

-format *formatString*AttributeName Display jobs with a custom format. See the *condor_q* man page **-format** option for details.

-l or -long Display job ads in long format.

-match *number* Limit the number of jobs displayed to *number*.

-name *schedd-name* When Quill is enabled, query job ClassAds from the named *condor_schedd* daemon, not the default *condor_schedd* daemon.

-xml Display job ClassAds in xml format. The xml format is fully defined at http://www.cs.wisc.edu/condor/classad/refman/.

Examples

To see all historical jobs since April 1, 2005 at 1pm,

```
%condor_history -completedsince '04/01/2005 13:00'
```

Exit Status

condor_history will exit with a status value of 0 (zero) upon success, and it will exit with the value 1 (one) upon failure.

Author

Condor Team, University of Wisconsin–Madison

Copyright

condor_hold

put jobs in the queue into the hold state

Synopsis

condor_hold [**-help** | **-version**]

condor_hold [**-debug**] [**-pool** *centralmanagerhostname[:portnumber]* | **-name** *scheddname*]| [**-addr** *"<a.b.c.d:port>"*] *cluster*. . .| *cluster.process*. . .| *user*. . . | **-constraint** *expression* . . .

condor_hold [**-debug**] [**-pool** *centralmanagerhostname[:portnumber]* | **-name** *scheddname*]| [**-addr** *"<a.b.c.d:port>"*] **-all**

Description

condor_hold places jobs from the Condor job queue in the hold state. If the **-name** option is specified, the named *condor_schedd* is targeted for processing. Otherwise, the local *condor_schedd* is targeted. The jobs to be held are identified by one or more job identifiers, as described below. For any given job, only the owner of the job or one of the queue super users (defined by the QUEUE_SUPER_USERS macro) can place the job on hold.

A job in the hold state remains in the job queue, but the job will not run until released with *condor_release*.

A currently running job that is placed in the hold state by *condor_hold* is sent a hard kill signal. For a standard universe job, this means that the job is removed from the machine without allowing a checkpoint to be produced first.

Options

-help Display usage information

-version Display version information

-pool *centralmanagerhostname[:portnumber]* Specify a pool by giving the central manager's host name and an optional port number

-name *scheddname* Send the command to a machine identified by *scheddname*

-addr *"<**a.b.c.d:port**>"* Send the command to a machine located at *"<a.b.c.d:port>"*

-debug Causes debugging information to be sent to `stderr`, based on the value of the configuration variable `TOOL_DEBUG`

cluster Hold all jobs in the specified cluster

cluster.process Hold the specific job in the cluster

user Hold all jobs belonging to specified user

-constraint ***expression*** Hold all jobs which match the job ClassAd expression constraint (within quotation marks). Note that quotation marks must be escaped with the backslash characters for most shells.

-all Hold all the jobs in the queue

See Also

condor_release (on page 786)

Examples

To place on hold all jobs (of the user that issued the *condor_hold* command) that are not currently running:

```
% condor_hold -constraint "JobStatus!=2"
```

Multiple options within the same command cause the union of all jobs that meet either (or both) of the options to be placed in the hold state. Therefore, the command

```
% condor_hold Mary -constraint "JobStatus!=2"
```

places all of Mary's queued jobs into the hold state, and the constraint holds all queued jobs not currently running. It also sends a hard kill signal to any of Mary's jobs that are currently running. Note that the jobs specified by the constraint will also be Mary's jobs, if it is Mary that issues this example *condor_hold* command.

Exit Status

condor_hold will exit with a status value of 0 (zero) upon success, and it will exit with the value 1 (one) upon failure.

Author

Condor Team, University of Wisconsin–Madison

Copyright

condor_load_history

Read a Condor history file into a Quill database

Synopsis

condor_load_history **-f** *historyfilename* [**-name** *schedd-name jobqueue-birthdate*]

Description

condor_load_history reads a Condor history file, adding its information to a Quill database. The Quill database is located via configuration variables. The history file to read is defined by the required **-f** *historyfilename* argument.

The combination of a *condor_schedd* daemon's name together with its process creation date (the job queue's birthdate) define a unique identifier that may be attached to the Quill database with the **-name** option. The format of birthdate expected is exactly the first line of the `job_queue.log` file. The location of this file may be determined using

```
% condor_config_val spool
```

Be aware and expect that the reading and processing of a sizable history file may take a large amount of time.

Options

-name ***schedd-name jobqueue-birthdate*** The *schedd-name* and *jobqueue-birthdate* combine to form a unique name for the database. The expected values are the name of the *condor_schedd* daemon and the first line of the `job_queue.log` file, which gives a job queue creation time.

Exit Status

condor_load_history will exit with a status value of 0 (zero) upon success, and it will exit with the value 1 (one) upon failure.

Author

Condor Team, University of Wisconsin–Madison

Copyright

condor_master

The master Condor Daemon

Synopsis

condor_master

Description

This daemon is responsible for keeping all the rest of the Condor daemons running on each machine in your pool. It spawns the other daemons, and periodically checks to see if there are new binaries installed for any of them. If there are, the *condor_master* will restart the affected daemons. In addition, if any daemon crashes, the *condor_master* will send e-mail to the Condor Administrator of your pool and restart the daemon. The *condor_master* also supports various administrative commands that let you start, stop or reconfigure daemons remotely. The *condor_master* will run on every machine in your Condor pool, regardless of what functions each machine are performing.

Section 3.1.2 in the Administrator's Manual has more information about the *condor_master* and other Condor daemons. See Section 3.9.2 for documentation on command line arguments for *condor_master*.

The `DAEMON_LIST` configuration macro is used by the *condor_master* to provide a per-machine list of daemons that should be started and kept running. For daemons that are specified in the `DC_DAEMON_LIST` configuration macro, the *condor_master* daemon will spawn them automatically appending a *-f* argument. For those listed in `DAEMON_LIST`, but not in `DC_DAEMON_LIST`, there will be no *-f* argument.

Options

-n *name* Provides an alternate name for the *condor_master* to override that given by the `MASTER_NAME` configuration variable.

Author

Condor Team, University of Wisconsin–Madison

Copyright

condor_master_off

Shutdown Condor and the *condor_master*

Synopsis

condor_master_off [**-help**] [**-version**] [**hostname ...**]

Description

condor_master_off no longer exists.

General Remarks

condor_master_off no longer exists as a Condor command. Instead, use

```
condor_off -master
```

to accomplish this task.

See Also

See the *condor_off* manual page.

Author

Condor Team, University of Wisconsin–Madison

Copyright

condor_off

Shutdown Condor daemons

Synopsis

condor_off [**-help** | **-version**]

condor_off [**-graceful** | **-fast** | **-peaceful**] [**-debug**] [**-pool** *centralmanagerhostname[:portnumber]*] [**-name** *hostname* | *hostname* | **-addr** *"<a.b.c.d:port>"* | *"<a.b.c.d:port>"* | **-constraint** *expression* | **-all**] [**-subsystem** *subsystemname*]

Description

condor_off shuts down a set of the Condor daemons running on a set of one or more machines. It does this cleanly so that checkpointable jobs may gracefully exit with minimal loss of work.

The command *condor_off* without any arguments will shut down all daemons except *condor_master*. The *condor_master* can then handle both local and remote requests to restart the other Condor daemons if need be. To restart Condor running on a machine, see the *condor_on* command.

With the **-subsystem** *master* option, *condor_off* will shut down all daemons including the *condor_master*. Specification using the **-subsystem** option will shut down only the specified daemon.

For security purposes (authentication and authorization), this command requires an administrator's level of access. See section 3.6.1 on page 315 for further explanation.

Options

-help Display usage information

-version Display version information

-graceful Gracefully shutdown daemons (the default)

-fast Quickly shutdown daemons

-peaceful Wait indefinitely for jobs to finish

-debug Causes debugging information to be sent to `stderr`, based on the value of the configuration variable `TOOL_DEBUG`

-pool ***centralmanagerhostname[:portnumber]*** Specify a pool by giving the central manager's host name and an optional port number

-name ***hostname*** Send the command to a machine identified by *hostname*

hostname Send the command to a machine identified by *hostname*

-addr *"<**a.b.c.d:port**>"* Send the command to a machine's master located at *"<a.b.c.d:port>"*

*"<**a.b.c.d:port**>"* Send the command to a machine located at *"<a.b.c.d:port>"*

-constraint ***expression*** Apply this command only to machines matching the given ClassAd *expression*

-all Send the command to all machines in the pool

-subsystem ***subsystemname*** Send the command to the named subsystem. Without this option, the command is sent to the *condor_master* daemon.

Exit Status

condor_off will exit with a status value of 0 (zero) upon success, and it will exit with the value 1 (one) upon failure.

Examples

To shut down all daemons (other than *condor_master*) on the local host:

```
% condor_off
```

To shut down only the *condor_collector* on three named machines:

```
% condor_off  cinnamon cloves vanilla -subsystem collector
```

To shut down daemons within a pool of machines other than the local pool, use the **-pool** option. The argument is the name of the central manager for the pool. Note that one or more machines within the pool must be specified as the targets for the command. This command shuts down all daemons except the *condor_master* on the single machine named **cae17** within the pool of machines that has **condor.cae.wisc.edu** as its central manager:

```
% condor_off  -pool condor.cae.wisc.edu -name cae17
```

Author

Condor Team, University of Wisconsin–Madison

Copyright

condor_on

Start up Condor daemons

Synopsis

condor_on [**-help** | **-version**]

condor_on [**-debug**] [**-pool** *centralmanagerhostname[:portnumber]*] [**-name** *hostname* | *hostname* | **-addr** *"<a.b.c.d:port>"* | *"<a.b.c.d:port>"* | **-constraint** *expression* | **-all**] [**-subsystem** *subsystemname*]

Description

condor_on starts up a set of the Condor daemons on a set of machines. This command assumes that the *condor_master* is already running on the machine. If this is not the case, *condor_on* will fail complaining that it cannot find the address of the master. The command *condor_on* with no arguments or with the **-subsystem** *master* option will tell the *condor_master* to start up the Condor daemons specified in the configuration variable `DAEMON_LIST`. If a daemon other than the *condor_master* is specified with the **-subsystem** option, *condor_on* starts up only that daemon.

This command cannot be used to start up the *condor_master* daemon.

For security purposes (authentication and authorization), this command requires an administrator's level of access. See section 3.6.1 on page 315 for further explanation.

Options

-help Display usage information

-version Display version information

-debug Causes debugging information to be sent to `stderr`, based on the value of the configuration variable `TOOL_DEBUG`

-pool *centralmanagerhostname[:portnumber]* Specify a pool by giving the central manager's host name and an optional port number

-name *hostname* Send the command to a machine identified by *hostname*

hostname Send the command to a machine identified by *hostname*

-addr *"<**a.b.c.d:port**>"* Send the command to a machine's master located at *"<a.b.c.d:port>"*

*"<**a.b.c.d:port**>"* Send the command to a machine located at *"<a.b.c.d:port>"*

-constraint ***expression*** Apply this command only to machines matching the given ClassAd *expression*

-all Send the command to all machines in the pool

-subsystem ***subsystemname*** Send the command to the named subsystem. Without this option, the command is sent to the *condor_master* daemon.

Exit Status

condor_on will exit with a status value of 0 (zero) upon success, and it will exit with the value 1 (one) upon failure.

Examples

To begin running all daemons (other than *condor_master*) given in the configuration variable `DAEMON_LIST` on the local host:

```
% condor_on
```

To start up only the *condor_negotiator* on two named machines:

```
% condor_on  robin cardinal -subsystem negotiator
```

To start up only a daemon within a pool of machines other than the local pool, use the **-pool** option. The argument is the name of the central manager for the pool. Note that one or more machines within the pool must be specified as the targets for the command. This command starts up only the *condor_schedd* daemon on the single machine named **cae17** within the pool of machines that has **condor.cae.wisc.edu** as its central manager:

```
% condor_on -pool condor.cae.wisc.edu -name cae17 -subsystem schedd
```

Author

Condor Team, University of Wisconsin–Madison

Copyright

condor_power

send packet intended to wake a machine from a low power state

Synopsis

condor_power [**-h**]

condor_power [**-d**] [**-m** *MACaddress*] [**-s** *subnet*] [*ClassAdFile*]

Description

condor_power sends one UDP Wake on LAN (WOL) packet to a machine specified either by command line arguments or by the contents of a machine ClassAd. The machine ClassAd may be in a file, where the file name specified by the optional argument *ClassAdFile* is given on the command line. With no command line arguments to specify the machine, and no file specified, *condor_power* quietly presumes that standard input is the file source which will specify the machine ClassAd that includes the public IP address and subnet of the machine.

condor_power needs a complete specification of the machine to be successful. If a MAC address is provided on the command line, but no subnet is given, then the default value for the subnet is used. If a subnet is provided on the command line, but no MAC address is given, then *condor_power* falls back to taking its information in the form of the machine ClassAd as provided in a file or on standard input. Note that this case implies that the command line specification of the subnet is ignored.

Options

-h Print usage information and exit.

-d Enable debugging messages.

-m *MACaddress* Specify the MAC address in the standard format of six groups of two hexadecimal digits separated by colons.

-s *subnet* Specify the subnet in the standard form of an IP address. Without this option, the default subnet used will be 255.255.255.255, causing a broadcast.

Exit Status

condor_power will exit with a status value of 0 (zero) upon success, and it will exit with the value 1 (one) upon failure.

Author

Condor Team, University of Wisconsin–Madison

Copyright

condor_preen

remove extraneous files from Condor directories

Synopsis

condor_preen [**-mail**] [**-remove**] [**-verbose**]

Description

condor_preen examines the directories belonging to Condor, and removes extraneous files and directories which may be left over from Condor processes which terminated abnormally either due to internal errors or a system crash. The directories checked are the `LOG`, `EXECUTE`, and `SPOOL` directories as defined in the Condor configuration files. *condor_preen* is intended to be run as user `root` or user `condor` periodically as a backup method to ensure reasonable file system cleanliness in the face of errors. This is done automatically by default by the *condor_master* daemon. It may also be explicitly invoked on an as needed basis.

When *condor_preen* cleans the `SPOOL` directory, it always leaves behind the files specified in the configuration variable `VALID_SPOOL_FILES` as given by the configuration. For the `LOG` directory, the only files removed or reported are those listed within the configuration variable `INVALID_LOG_FILES` list. The reason for this difference is that, in general, the files in the `LOG` directory ought to be left alone, with few exceptions. An example of exceptions are core files. As there are new log files introduced regularly, it is less effort to specify those that ought to be removed than those that are not to be removed.

Options

-mail Send mail to the user defined in the `PREEN_ADMIN` configuration variable, instead of writing to the standard output.

-remove Remove the offending files and directories rather than reporting on them.

-verbose List all files found in the Condor directories, even those which are not considered extraneous.

Exit Status

condor_preen will exit with a status value of 0 (zero) upon success, and it will exit with the value 1 (one) upon failure.

Author

Condor Team, University of Wisconsin–Madison

Copyright

condor_prio

change priority of jobs in the condor queue

Synopsis

condor_prio [**-p** *priority*] [**+** | **-** *value*] [**-n** *schedd_name*] [**-pool** *pool_name*] *cluster* | *cluster.process* | *username* | *-a*

Description

condor_prio changes the priority of one or more jobs in the condor queue. If a cluster_id and a process_id are both specified, *condor_prio* attempts to change the priority of the specified process. If a cluster_id is specified without a process_id, *condor_prio* attempts to change priority for all processes belonging to the specified cluster. If a username is specified, *condor_prio* attempts to change priority of all jobs belonging to that user. If the -a flag is set, *condor_prio* attempts to change priority of all jobs in the condor queue. The user must specify a priority adjustment or new priority. If the -p option is specified, the priority of the job(s) are set to the next argument. The user can also adjust the priority by supplying a + or - immediately followed by a digit. The priority of a job can be any integer, with higher numbers corresponding to greater priority. Only the owner of a job or the super user can change the priority for it.

The priority changed by *condor_prio* is only compared to the priority of other jobs owned by the same user and submitted from the same machine. See the "Condor Users and Administrators Manual" for further details on Condor's priority scheme.

Options

-p *priority* Set priority to the specified value

+ | **-** ***value*** Change priority by the specified value

-n *schedd_name* Change priority of jobs queued at the specified schedd in the local pool

-pool *pool_name* Change priority of jobs queued at the specified schedd in the specified pool

-a Change priority of all the jobs in the queue

Exit Status

condor_prio will exit with a status value of 0 (zero) upon success, and it will exit with the value 1 (one) upon failure.

Author

Condor Team, University of Wisconsin–Madison

Copyright

condor_procd

Track and manage process families

Synopsis

***condor_procd* -h**

***condor_procd* -A** *address-file* [**options**]

Description

condor_procd tracks and manages process families on behalf of the Condor daemons. It may track families of PIDs via relationships such as: direct parent/child, environment variables, UID, and supplementary group IDs. Management of the PID families include

- registering new families or new members of existing families
- getting usage information
- signaling families for operations such as suspension, continuing, or killing the family
- getting a snapshot of the tree of families

In a regular Condor installation, this program is not intended to be used or executed by any human.

The required argument, **-A** *address-file*, is the path and file name of the address file which is the named pipe that clients must use to speak with the *condor_procd*.

Options

-h Print out usage information and exit.

-D Wait for the debugger. Initially sleep 30 seconds before beginning normal function.

-C *principal* The *principal* is the UID of the owner of the named pipe that clients must use to speak to the *condor_procd*.

-L *log-file* A file the *condor_procd* will use to write logging information.

-E When specified, another tool such as the *procd_ctl* tool must allocate the GID associated with a process. When this option is *not* specified, the *condor_procd* will allocate the GID itself.

-P *PID* If not specified, the *condor_procd* will use the *condor_procd*'s parent, which may not be PID 1 on Unix, as the parent of the *condor_procd* and the root of the tracking family. When not specified, if the *condor_procd*'s parent PID dies, the *condor_procd* exits. When specified, the *condor_procd* will track this *PID* family in question and not also exit if the PID exits.

-S *seconds* The maximum number of seconds the *condor_procd* will wait between taking snapshots of the tree of families. Different clients to the *condor_procd* can specify different snapshot times. The quickest snapshot time is the one performed by the *condor_procd*. When this option is not specified, a default value of 60 seconds is used.

-G *min-gid max-gid* If the **-E** option is *not* specified, then track process families using a self-allocated, free GID out of the inclusive range specified by *min-gid* and *max-gid*. This means that if a new process shows up using a previously known GID, the new process will automatically associate into the process family assigned that GID. If the **-E** option *is* specified, then instead of self-allocating the GID, the *procd_ctl* tool must be used to associate the GID with the PID root of the family. The associated GID must still be in the range specified. This is a Linux-only feature.

-K *windows-softkill-binary* This is the path and executable name of the *condor_softkill.exe* binary. It is used to send softkill signals to process families. This is a Windows-only feature.

-I *glexec-kill-path glexec-path* Specifies, with *glexec-kill-path*, the path and executable name of a binary used to send a signal to a PID. The *glexec* binary, specified by *glexec-path*, executes the program specified with *glexec-kill-path* under the right privileges to send the signal.

General Remarks

This program may be used in a stand alone mode, independent of Condor, to track process families. The programs *procd_ctl* and *gidd_alloc* are used with the *condor_procd* in stand alone mode to interact with the daemon and to inquire about certain state of running processes on the machine, respectively.

Exit Status

condor_procd will exit with a status value of 0 (zero) upon success, and it will exit with the value 1 (one) upon failure.

Author

Condor Team, University of Wisconsin–Madison

Copyright

condor_q

Display information about jobs in queue

Synopsis

condor_q [**-help**]

condor_q [**-debug**] [**-global**] [**-submitter** *submitter*] [**-name** *name*] [**-pool** *centralmanagerhostname[:portnumber]*] [**-analyze**] [**-run**] [**-hold**] [**-globus**] [**-goodput**] [**-io**] [**-dag**] [**-long**] [**-xml**] [**-attributes** *Attr1 [,Attr2 . . .]*] [**-format** *fmt attr*] [**-cputime**] [**-currentrun**] [**-avgqueuetime**] [**-jobads** *file*] [**-machineads** *file*] [**-direct** *rdbms* | *schedd*] [*{cluster* | *cluster.process* | *owner* | ***-constraint*** *expression . . .}*]

Description

condor_q displays information about jobs in the Condor job queue. By default, *condor_q* queries the local job queue but this behavior may be modified by specifying:

- the **-global** option, which queries all job queues in the pool
- a schedd name with the **-name** option, which causes the queue of the named schedd to be queried
- a submitter with the **-submitter** option, which causes all queues of the named submitter to be queried

To restrict the display to jobs of interest, a list of zero or more restrictions may be supplied. Each restriction may be one of:

- a *cluster* and a *process* matches jobs which belong to the specified cluster and have the specified process number
- a *cluster* without a *process* matches all jobs belonging to the specified cluster
- a *owner* matches all jobs owned by the specified owner
- a **-constraint** *expression* which matches all jobs that satisfy the specified ClassAd expression. (See section 4.1 for a discussion of ClassAd expressions.)

If no *owner* restrictions are present in the list, the job matches the restriction list if it matches at least one restriction in the list. If *owner* restrictions are present, the job matches the list if it matches one of the *owner* restrictions *and* at least one non-owner restriction.

If the **-long** option is specified, *condor_q* displays a long description of the queried jobs by printing the entire job ClassAd. The attributes of the job ClassAd may be displayed by means of the **-format** option, which displays attributes with a `printf(3)` format. Multiple **-format** options may be specified in the option list to display several attributes of the job. If neither **-long** or **-format** are specified, *condor_q* displays a a one line summary of information as follows:

ID The cluster/process id of the condor job.

OWNER The owner of the job.

SUBMITTED The month, day, hour, and minute the job was submitted to the queue.

RUN_TIME Wall-clock time accumulated by the job to date in days, hours, minutes, and seconds.

ST Current status of the job, which varies somewhat according to the job universe and the timing of updates. H = on hold, R = running, I = idle (waiting for a machine to execute on), C = completed, X = removed, and > = transferring output.

PRI User specified priority of the job, ranges from -20 to +20, with higher numbers corresponding to greater priority.

SIZE The virtual image size of the executable in megabytes.

CMD The name of the executable.

If the **-dag** option is specified, the OWNER column is replaced with NODENAME for jobs started by Condor DAGMan.

If the **-run** option is specified, the ST, PRI, SIZE, and CMD columns are replaced with:

HOST(S) The host where the job is running.

If the **-globus** option is specified, the ST, PRI, SIZE, and CMD columns are replaced with:

STATUS The state that Condor believes the job is in. Possible values are

PENDING The job is waiting for resources to become available in order to run.

ACTIVE The job has received resources, and the application is executing.

FAILED The job terminated before completion because of an error, user-triggered cancel, or system-triggered cancel.

DONE The job completed successfully.

SUSPENDED The job has been suspended. Resources which were allocated for this job may have been released due to a scheduler-specific reason.

UNSUBMITTED The job has not been submitted to the scheduler yet, pending the reception of the GLOBUS_GRAM_PROTOCOL_JOB_SIGNAL_COMMIT_REQUEST signal from a client.

STAGE_IN The job manager is staging in files, in order to run the job.

STAGE_OUT The job manager is staging out files generated by the job.

UNKNOWN

MANAGER A guess at what remote batch system is running the job. It is a guess, because Condor looks at the Globus jobmanager contact string to attempt identification. If the value is fork, the job is running on the remote host without a jobmanager. Values may also be condor, lsf, or pbs.

HOST The host to which the job was submitted.

EXECUTABLE The job as specified as the executable in the submit description file.

If the **-goodput** option is specified, the ST, PRI, SIZE, and CMD columns are replaced with:

GOODPUT The percentage of RUN_TIME for this job which has been saved in a checkpoint. A low GOODPUT value indicates that the job is failing to checkpoint. If a job has not yet attempted a checkpoint, this column contains `[?????]`.

CPU_UTIL The ratio of CPU_TIME to RUN_TIME for checkpointed work. A low CPU_UTIL indicates that the job is not running efficiently, perhaps because it is I/O bound or because the job requires more memory than available on the remote workstations. If the job has not (yet) checkpointed, this column contains `[??????]`.

Mb/s The network usage of this job, in Megabits per second of run-time.

If the **-io** option is specified, the ST, PRI, SIZE, and CMD columns are replaced with:

READ The total number of bytes the application has read from files and sockets.

WRITE The total number of bytes the application has written to files and sockets.

SEEK The total number of seek operations the application has performed on files.

XPUT The effective throughput (average bytes read and written per second) from the application's point of view.

BUFSIZE The maximum number of bytes to be buffered per file.

BLOCKSIZE The desired block size for large data transfers.

These fields are updated when a job produces a checkpoint or completes. If a job has not yet produced a checkpoint, this information is not available.

If the **-cputime** option is specified, the RUN_TIME column is replaced with:

CPU_TIME The remote CPU time accumulated by the job to date (which has been stored in a checkpoint) in days, hours, minutes, and seconds. (If the job is currently running, time accumulated during the current run is *not* shown. If the job has not produced a checkpoint, this column contains 0+00:00:00.)

The *-analyze* option may be used to determine why certain jobs are not running by performing an analysis on a per machine basis for each machine in the pool. The reasons may vary among failed constraints, insufficient priority, resource owner preferences and prevention of preemption by the `PREEMPTION_REQUIREMENTS` expression. If the *-long* option is specified along with the *-analyze* option, the reason for failure is displayed on a per machine basis.

Options

-help Get a brief description of the supported options

-global Get queues of all the submitters in the system

-debug Causes debugging information to be sent to `stderr`, based on the value of the configuration variable `TOOL_DEBUG`

-submitter *submitter* List jobs of specific submitter from all the queues in the pool

-pool *centralmanagerhostname[:portnumber]* Use the *centralmanagerhostname* as the central manager to locate schedds. (The default is the `COLLECTOR_HOST` specified in the configuration file.

-analyze Perform an analysis to determine how many resources are available to run the requested jobs. These results are only meaningful for jobs using Condor's matchmaker. This option is never meaningful for Scheduler universe jobs and only meaningful for grid universe jobs doing matchmaking.

-run Get information about running jobs.

-hold Get information about jobs in the hold state. Also displays the time the job was placed into the hold state and the reason why the job was placed in the hold state.

-globus Get information only about jobs submitted to grid resources described as **gt2** or **gt4**.

-goodput Display job goodput statistics.

-io Display job input/output summaries.

-dag Display DAG jobs under their DAGMan.

-name *name* Show only the job queue of the named schedd

-long Display job ads in long format

-xml Display job ads in xml format. The xml format is fully defined at http://www.cs.wisc.edu/condor/classad/refman/.

-attributes *Attr1 [,Attr2* . . .*]* Explicitly list the attributes (by name, and in a comma separated list) which should be displayed when using the **-xml** or **-long** options. Limiting the number of attributes increases the efficiency of the query.

-format *fmt attr* Display attribute or expression *attr* in format *fmt*. To display the attribute or expression the format must contain a single `printf(3)` style conversion specifier. Attributes must be from the job ClassAd. Expressions are ClassAd expressions and may refer to attributes in the job ClassAd. If the attribute is not present in a given ClassAd and cannot be parsed as an expression, then the format option will be silently skipped. The conversion specifier must match the type of the attribute or expression. %s is suitable for strings such as `Owner`, %d for integers such as `ClusterId`, and %f for floating point numbers such as `RemoteWallClockTime`. An incorrect format will result in undefined behavior. Do not use more than one conversion specifier in a given format. More than one conversion specifier will result in undefined behavior. To output multiple attributes repeat the **-format** option once for each desired attribute. Like `printf(3)` style formats, you can include other text that will be reproduced directly. You can specify a format without any conversion specifiers but you must still give attribute. You can include \n to specify a line break.

-cputime Instead of wall-clock allocation time (RUN_TIME), display remote CPU time accumulated by the job to date in days, hours, minutes, and seconds. (If the job is currently running, time accumulated during the current run is *not* shown.)

-currentrun Normally, RUN_TIME contains all the time accumulated during the current run plus all previous runs. If this option is specified, RUN_TIME only displays the time accumulated so far on this current run.

-avgqueuetime Display the average of time spent in the queue, considering all jobs not completed (those that do not have `JobStatus == 4` or `JobStatus == 3`.

-jobads *file* Display jobs from a list of ClassAds from a file, instead of the real ClassAds from the *condor_schedd* daemon. This is most useful for debugging purposes. The ClassAds appear as if *condor_q* **-l** is used with the header stripped out.

-machineads *file* When doing analysis, use the machine ads from the file instead of the ones from the *condor_collector* daemon. This is most useful for debugging purposes. The ClassAds appear as if *condor_status* **-l** is used.

-direct *rdbms* | *schedd* When the use of Quill is enabled, this option allows a direct query to either the rdbms or the *condor_schedd* daemon for the requested queue information. It also prevents the queue location discovery algorithm from failing over to alternate sources of information for the queue in case of error. It is useful for debugging an installation of Quill. One of the strings *rdbms* or *schedd* is required with this option.

Restriction list The restriction list may have zero or more items, each of which may be:

cluster match all jobs belonging to cluster

cluster.proc match all jobs belonging to cluster with a process number of *proc*

-constraint *expression* match all jobs which match the ClassAd expression constraint

A job matches the restriction list if it matches any restriction in the list Additionally, if *owner* restrictions are supplied, the job matches the list only if it also matches an *owner* restriction.

General Remarks

The default output from *condor_q* is formatted to be human readable, not script readable. In an effort to make the output fit within 80 characters, values in some fields might be truncated. Furthermore, the Condor Project can (and does) change the formatting of this default output as we see fit. Therefore, any script that is attempting to parse data from *condor_q* is strongly encouraged to use the **-format** option (described above, examples given below).

Although *-analyze* provides a very good first approximation, the analyzer cannot diagnose all possible situations because the analysis is based on instantaneous and local information. Therefore, there are some situations (such as when several submitters are contending for resources, or if the pool is rapidly changing state) which cannot be accurately diagnosed.

-goodput, *-cputime*, and *-io* are most useful for STANDARD universe jobs, since they rely on values computed when a job checkpoints.

Examples

The **-format** option provides a way to specify both the job attributes and formatting of those attributes. There must be only one conversion specification per **-format** option. As an example, to list only Jane Doe's jobs in the queue, choosing to print and format only the owner of the job, the command line arguments for the job, and the process ID of the job:

```
%condor_q -submitter jdoe -format "%s" Owner -format " %s " Args -format "ProcId = %d\n" ProcId
jdoe 16386 2800 ProcId = 0
jdoe 16386 3000 ProcId = 1
jdoe 16386 3200 ProcId = 2
jdoe 16386 3400 ProcId = 3
jdoe 16386 3600 ProcId = 4
jdoe 16386 4200 ProcId = 7
```

To display only the JobID's of Jane Doe's jobs you can use the following.

```
%condor_q -submitter jdoe -format "%d." ClusterId -format "%d\n" ProcId
27.0
27.1
27.2
27.3
27.4
27.7
```

An example that shows the difference (first set of output) between not using an option to *condor_q* and (second set of output) using the **-globus** option:

```
 ID      OWNER            SUBMITTED     RUN_TIME ST PRI SIZE CMD
 100.0   smith          12/11 13:20   0+00:00:02 R  0   0.0  sleep 10

1 jobs; 0 idle, 1 running, 0 held


 ID      OWNER          STATUS  MANAGER  HOST                EXECUTABLE
 100.0   smith         ACTIVE fork     grid.example.com       /bin/sleep
```

Exit Status

condor_q will exit with a status value of 0 (zero) upon success, and it will exit with the value 1 (one) upon failure.

Author

Condor Team, University of Wisconsin–Madison

Copyright

condor_qedit

modify job attributes

Synopsis

condor_qedit [**-debug**] [**-n** *schedd-name*] [**-pool** *pool-name*] *{cluster | cluster.proc | owner | -constraint constraint} attribute-name attribute-value . . .*

Description

condor_qedit modifies job ClassAd attributes of queued Condor jobs. The jobs are specified either by cluster number, job ID, owner, or by a ClassAd constraint expression. The *attribute-value* may be any ClassAd expression. String expressions must be surrounded by double quotes.

To ensure security and correctness, *condor_qedit* will not allow modification of the following ClassAd attributes:

- `Owner`
- `ClusterId`
- `ProcId`
- `MyType`
- `TargetType`
- `JobStatus`

Since `JobStatus` may not be changed with *condor_qedit*, use *condor_hold* to place a job in the hold state, and use *condor_release* to release a held job, instead of attempting to modify `JobStatus` directly.

If a job is currently running, modified attributes for that job will not affect the job until it restarts. As an example, for `PeriodicRemove` to affect when a currently running job will be removed from the queue, that job must first be evicted from a machine and returned to the queue. The same is true for other periodic expressions, such as `PeriodicHold` and `PeriodicRelease`.

Options

-debug Causes debugging information to be sent to `stderr`, based on the value of the configuration variable `TOOL_DEBUG`

-n *schedd-name* Modify job attributes in the queue of the specified schedd

-pool *pool-name* Modify job attributes in the queue of the schedd specified in the specified pool

Examples

```
% condor_qedit -name north.cs.wisc.edu -pool condor.cs.wisc.edu 249.0 answer 42
Set attribute "answer".
% condor_qedit -name perdita 1849.0 In '"myinput"'
Set attribute "In".
% condor_qedit jbasney NiceUser TRUE
Set attribute "NiceUser".
% condor_qedit -constraint 'JobUniverse == 1' Requirements '(Arch == "INTEL") && (OpSys == "SOLARIS26") &&
Set attribute "Requirements".
```

General Remarks

A job's ClassAd attributes may be viewed with

```
condor_q -long
```

Exit Status

condor_qedit will exit with a status value of 0 (zero) upon success, and it will exit with the value 1 (one) upon failure.

Author

Condor Team, University of Wisconsin–Madison

Copyright

condor_reconfig

Reconfigure Condor daemons

Synopsis

condor_reconfig [**-help** | **-version**]

condor_reconfig [**-debug**] [**-pool** *centralmanagerhostname[:portnumber]*] [**-name** *hostname* | *hostname* | **-addr** *"<a.b.c.d:port>"* | *"<a.b.c.d:port>"* | **-constraint** *expression* | **-all**] [**-subsystem** *subsystemname*]

Description

condor_reconfig reconfigures all of the Condor daemons in accordance with the current status of the Condor configuration file(s). Once reconfiguration is complete, the daemons will behave according to the policies stated in the configuration file(s). The main exception is with the `DAEMON_LIST` variable, which will only be updated if the *condor_restart* command is used. Other configuration variables that can only be changed if the Condor daemons are restarted are listed in section 3.3.1 on page 158. In general, *condor_reconfig* should be used when making changes to the configuration files, since it is faster and more efficient than restarting the daemons.

The command *condor_reconfig* with no arguments or with the **-subsystem** *master* option will cause the reconfiguration of the *condor_master* daemon and all the child processes of the *condor_master*.

For security purposes (authentication and authorization), this command requires an administrator's level of access. See section 3.6.1 on page 315 for further explanation.

Options

-help Display usage information

-version Display version information

-debug Causes debugging information to be sent to `stderr`, based on the value of the configuration variable `TOOL_DEBUG`

-pool ***centralmanagerhostname[:portnumber]*** Specify a pool by giving the central manager's host name and an optional port number

-name *hostname* Send the command to a machine identified by *hostname*

hostname Send the command to a machine identified by *hostname*

-addr *"<**a.b.c.d:port**>"* Send the command to a machine's master located at *"<a.b.c.d:port>"*

*"<**a.b.c.d:port**>"* Send the command to a machine located at *"<a.b.c.d:port>"*

-constraint *expression* Apply this command only to machines matching the given ClassAd *expression*

-all Send the command to all machines in the pool

-subsystem *subsystemname* Send the command to the named subsystem. Without this option, the command is sent to the *condor_master* daemon.

Exit Status

condor_reconfig will exit with a status value of 0 (zero) upon success, and it will exit with the value 1 (one) upon failure.

Examples

To reconfigure the *condor_master* and all its children on the local host:

```
% condor_reconfig
```

To reconfigure only the *condor_startd* on a named machine:

```
% condor_reconfig -name bluejay -subsystem startd
```

To reconfigure a machine within a pool other than the local pool, use the **-pool** option. The argument is the name of the central manager for the pool. Note that one or more machines within the pool must be specified as the targets for the command. This command reconfigures the single machine named **cae17** within the pool of machines that has **condor.cae.wisc.edu** as its central manager:

```
% condor_reconfig -pool condor.cae.wisc.edu -name cae17
```

Author

Condor Team, University of Wisconsin–Madison

Copyright

condor_reconfig_schedd

Reconfigure condor schedd

Synopsis

condor_reconfig_schedd [**-help**] [**-version**] [**hostname ...**]

Description

condor_reconfig_schedd no longer exists.

General Remarks

condor_reconfig_schedd no longer exists as a Condor command. Instead, use

```
condor_reconfig -schedd
```

to accomplish this task.

See Also

See the *condor_reconfig* manual page.

Author

Condor Team, University of Wisconsin–Madison

Copyright

condor_release

release held jobs in the Condor queue

Synopsis

condor_release [**-help** | **-version**]

condor_release [**-debug**] [**-pool** *centralmanagerhostname[:portnumber]* | **-name** *scheddname*]| [**-addr** *"<a.b.c.d:port>"*] *cluster*. . .| *cluster.process*. . .| *user*. . . | **-constraint** *expression* . . .

condor_release [**-debug**] [**-pool** *centralmanagerhostname[:portnumber]* | **-name** *scheddname*]| [**-addr** *"<a.b.c.d:port>"*] **-all**

Description

condor_release releases jobs from the Condor job queue that were previously placed in hold state. If the **-name** option is specified, the named *condor_schedd* is targeted for processing. Otherwise, the local *condor_schedd* is targeted. The jobs to be released are identified by one or more job identifiers, as described below. For any given job, only the owner of the job or one of the queue super users (defined by the QUEUE_SUPER_USERS macro) can release the job.

Options

-help Display usage information

-version Display version information

-pool *centralmanagerhostname[:portnumber]* Specify a pool by giving the central manager's host name and an optional port number

-name *scheddname* Send the command to a machine identified by *scheddname*

-addr *"<a.b.c.d:port>"* Send the command to a machine located at *"<a.b.c.d:port>"*

-debug Causes debugging information to be sent to `stderr`, based on the value of the configuration variable TOOL_DEBUG

cluster Release all jobs in the specified cluster

cluster.process Release the specific job in the cluster

user Release jobs belonging to specified user

-constraint *expression* Release all jobs which match the job ClassAd expression constraint

-all Release all the jobs in the queue

See Also

condor_hold (on page 749)

Examples

To release all of the jobs of a user named Mary:

```
% condor_release Mary
```

Exit Status

condor_release will exit with a status value of 0 (zero) upon success, and it will exit with the value 1 (one) upon failure.

Author

Condor Team, University of Wisconsin–Madison

Copyright

condor_reschedule

Update scheduling information to the central manager

Synopsis

condor_reschedule [**-help** | **-version**]

condor_reschedule [**-debug**] [**-pool** *centralmanagerhostname[:portnumber]*] [**-name** *hostname* | *hostname* | **-addr** *"<a.b.c.d:port>"* | *"<a.b.c.d:port>"* | **-constraint** *expression* | **-all**]

Description

condor_reschedule updates the information about a set of machines' resources and jobs to the central manager. This command is used to force an update before viewing the current status of a machine. Viewing the status of a machine is done with the *condor_status* command. *condor_reschedule* also starts a new negotiation cycle between resource owners and resource providers on the central managers, so that jobs can be matched with machines right away. This can be useful in situations where the time between negotiation cycles is somewhat long, and an administrator wants to see if a job in the queue will get matched without waiting for the next negotiation cycle.

A new negotiation cycle cannot occur more frequently than every 20 seconds. Requests for new negotiation cycle within that 20 second window will be deferred until 20 seconds have passed since that last cycle.

Options

-help Display usage information

-version Display version information

-debug Causes debugging information to be sent to `stderr`, based on the value of the configuration variable `TOOL_DEBUG`

-pool *centralmanagerhostname[:portnumber]* Specify a pool by giving the central manager's host name and an optional port number

-name *hostname* Send the command to a machine identified by *hostname*

hostname Send the command to a machine identified by *hostname*

-addr *"<**a.b.c.d:port**>"* Send the command to a machine's master located at *"<a.b.c.d:port>"*

*"<**a.b.c.d:port**>"* Send the command to a machine located at *"<a.b.c.d:port>"*

-constraint ***expression*** Apply this command only to machines matching the given ClassAd *expression*

-all Send the command to all machines in the pool

Exit Status

condor_reschedule will exit with a status value of 0 (zero) upon success, and it will exit with the value 1 (one) upon failure.

Examples

To update the information on three named machines:

```
% condor_reschedule robin cardinal bluejay
```

To reschedule on a machine within a pool other than the local pool, use the **-pool** option. The argument is the name of the central manager for the pool. Note that one or more machines within the pool must be specified as the targets for the command. This command reschedules the single machine named **cae17** within the pool of machines that has **condor.cae.wisc.edu** as its central manager:

```
% condor_reschedule -pool condor.cae.wisc.edu -name cae17
```

Author

Condor Team, University of Wisconsin–Madison

Copyright

condor_restart

Restart a set of Condor daemons

Synopsis

condor_restart [**-help** | **-version**]

condor_restart [**-debug**] [**-graceful** | **-fast** | **-peaceful**] [**-pool** *centralmanagerhostname[:portnumber]*] [**-name** *hostname* | *hostname* | **-addr** *"<a.b.c.d:port>"* | *"<a.b.c.d:port>"* | **-constraint** *expression* | **-all**] [**-subsystem** *subsystemname*]

Description

condor_restart restarts a set of Condor daemons on a set of machines. The daemons will be put into a consistent state, killed, and then invoked anew.

If, for example, the *condor_master* needs to be restarted again with a fresh state, this is the command that should be used to do so. If the `DAEMON_LIST` variable in the configuration file has been changed, this command is used to restart the *condor_master* in order to see this change. The *condor_reconfigure* command cannot be used in the case where the `DAEMON_LIST` expression changes.

The command *condor_restart* with no arguments or with the **-subsystem** *master* option will safely shut down all running jobs and all submitted jobs from the machine(s) being restarted, then shut down all the child daemons of the *condor_master*, and then restart the *condor_master*. This, in turn, will allow the *condor_master* to start up other daemons as specified in the `DAEMON_LIST` configuration file entry.

For security purposes (authentication and authorization), this command requires an administrator's level of access. See section 3.6.1 on page 315 for further explanation.

Options

-help Display usage information

-version Display version information

-debug Causes debugging information to be sent to `stderr`, based on the value of the configuration variable `TOOL_DEBUG`

-graceful Gracefully shutdown daemons (the default) before restarting them

-fast Quickly shutdown daemons before restarting them

-peaceful Wait indefinitely for jobs to finish before shutting down daemons, prior to restarting them

-pool *centralmanagerhostname[:portnumber]* Specify a pool by giving the central manager's host name and an optional port number

-name *hostname* Send the command to a machine identified by *hostname*

hostname Send the command to a machine identified by *hostname*

-addr *"<**a.b.c.d:port**>"* Send the command to a machine's master located at *"<a.b.c.d:port>"*

*"<**a.b.c.d:port**>"* Send the command to a machine located at *"<a.b.c.d:port>"*

-constraint *expression* Apply this command only to machines matching the given ClassAd *expression*

-all Send the command to all machines in the pool

-subsystem *subsystemname* Send the command to the named subsystem. Without this option, the command is sent to the *condor_master* daemon.

Exit Status

condor_restart will exit with a status value of 0 (zero) upon success, and it will exit with the value 1 (one) upon failure.

Examples

To restart the *condor_master* and all its children on the local host:

```
% condor_restart
```

To restart only the *condor_startd* on a named machine:

```
% condor_restart -name bluejay -subsystem startd
```

To restart a machine within a pool other than the local pool, use the **-pool** option. The argument is the name of the central manager for the pool. Note that one or more machines within the pool must be specified as the targets for the command. This command restarts the single machine named **cae17** within the pool of machines that has **condor.cae.wisc.edu** as its central manager:

```
% condor_restart -pool condor.cae.wisc.edu -name cae17
```

Author

Condor Team, University of Wisconsin–Madison

Copyright

condor_rm

remove jobs from the Condor queue

Synopsis

condor_rm [**-help** | **-version**]

condor_rm [**-debug**] [**-forcex**] [**-pool** *centralmanagerhostname[:portnumber]* | **-name** *scheddname*] [**-addr** *"<a.b.c.d:port>"*] *cluster...* | *cluster.process...* | *user...* | **-constraint** *expression ...*

condor_rm [**-debug**] [**-pool** *centralmanagerhostname[:portnumber]* | **-name** *scheddname*] [**-addr** *"<a.b.c.d:port>"*] **-all**

Description

condor_rm removes one or more jobs from the Condor job queue. If the **-name** option is specified, the named *condor_schedd* is targeted for processing. Otherwise, the local *condor_schedd* is targeted. The jobs to be removed are identified by one or more job identifiers, as described below. For any given job, only the owner of the job or one of the queue super users (defined by the QUEUE_SUPER_USERS macro) can remove the job.

When removing a grid job, the job may remain in the "X" state for a very long time. This is normal, as Condor is attempting to communicate with the remote scheduling system, ensuring that the job has been properly cleaned up. If it takes too long, or in rare circumstances is never removed, the job may be forced to leave the job queue by using the **-forcex** option. This forcibly removes jobs that are in the "X" state without attempting to finish any clean up at the remote scheduler.

Options

-help Display usage information

-version Display version information

-pool ***centralmanagerhostname[:portnumber]*** Specify a pool by giving the central manager's host name and an optional port number

-name ***scheddname*** Send the command to a machine identified by *scheddname*

-addr *"<**a.b.c.d:port**>"* Send the command to a machine located at *"<a.b.c.d:port>"*

-debug Causes debugging information to be sent to `stderr`, based on the value of the configuration variable `TOOL_DEBUG`

-forcex Force the immediate local removal of jobs in the 'X' state (only affects jobs already being removed)

cluster Remove all jobs in the specified cluster

cluster.process Remove the specific job in the cluster

user Remove jobs belonging to specified user

-constraint *expression* Remove all jobs which match the job ClassAd expression constraint

-all Remove all the jobs in the queue

General Remarks

Use the *-forcex* argument with caution, as it will remove jobs from the local queue immediately, but can orphan parts of the job that are running remotely and have not yet been stopped or removed.

Examples

For a user to remove all their jobs that are not currently running:

```
% condor_rm -constraint 'JobStatus =!= 2'
```

Exit Status

condor_rm will exit with a status value of 0 (zero) upon success, and it will exit with the value 1 (one) upon failure.

Author

Condor Team, University of Wisconsin–Madison

Copyright

condor_router_history

Display the history for routed jobs

Synopsis

condor_router_history [- -**h**]

condor_router_history [- -**show_records**] [- -**show_iwd**] [- -**age** *days*] [- -**days** *days*] [- -**start** *"YYYY-MM-DD HH:MM"*]

Description

condor_router_history summarizes statistics for routed jobs over the previous 24 hours. With no command line options, statistics for run time, number of jobs completed, and number of jobs aborted are listed per route (site).

Options

—h Display usage information and exit.

—show_records Displays individual records in addition to the summary.

—show_iwd Include working directory in displayed records.

—age *days* Set the ending time of the summary to be *days* days ago.

—days *days* Set the number of days to summarize.

—start *"YYYY-MM-DD HH:MM"* Set the start time of the summary.

Exit Status

condor_router_history will exit with a status of 0 (zero) upon success, and non-zero otherwise.

Author

Condor Team, University of Wisconsin–Madison

Copyright

condor_router_q

Display information about routed jobs in the queue

Synopsis

condor_router_q [**-S**] [**-R**] [**-I**] [**-H**] [**-route** *name*] [**-idle**] [**-held**] [**-constraint** *X*] [**condor_q options**]

Description

condor_router_q displays information about jobs managed by the *condor_job_router* that are in the Condor job queue. The functionality of this tool is that of *condor_q*, with additional options specialized for routed jobs. Therefore, any of the options for *condor_q* may also be used with *condor_router_q*.

Options

-S Summarize the state of the jobs on each route.

-R Summarize the running jobs on each route.

-I Summarize the idle jobs on each route.

-H Summarize the held jobs on each route.

-route *name* Display only the jobs on the route identified by *name*.

-idle Display only the idle jobs.

-held Display only the held jobs.

-constraint *X* Display only the jobs matching constraint *X*.

Exit Status

condor_router_q will exit with a status of 0 (zero) upon success, and non-zero otherwise.

Author

Condor Team, University of Wisconsin–Madison

Copyright

condor_run

Submit a shell command-line as a Condor job

Synopsis

condor_run [**-u** *universe*] *"shell command"*

Description

condor_run bundles a shell command line into a Condor job and submits the job. The *condor_run* command waits for the Condor job to complete, writes the job's output to the terminal, and exits with the exit status of the Condor job. No output appears until the job completes.

Enclose the shell command line in double quote marks, so it may be passed to *condor_run* without modification. *condor_run* will not read input from the terminal while the job executes. If the shell command line requires input, redirect the input from a file, as illustrated by the example

```
% condor_run "myprog < input.data"
```

condor_run jobs rely on a shared file system for access to any necessary input files. The current working directory of the job must be accessible to the machine within the Condor pool where the job runs.

Specialized environment variables may be used to specify requirements for the machine where the job may run.

CONDOR_ARCH Specifies the architecture of the required platform. Values will be the same as the `Arch` machine ClassAd attribute.

CONDOR_OPSYS Specifies the operating system of the required platform. Values will be the same as the `OpSys` machine ClassAd attribute.

CONDOR_REQUIREMENTS Specifies any additional requirements for the Condor job. It is recommended that the value defined for `CONDOR_REQUIREMENTS` be enclosed in parenthesis.

When one or more of these environment variables is specified, the job is submitted with:

```
Requirements = $CONDOR_REQUIREMENTS && Arch == $CONDOR_ARCH && \
   OpSys == $CONDOR_OPSYS
```

Without these environment variables, the job receives the default requirements expression, which requests a machine of the same platform as the machine on which *condor_run* is executed.

All environment variables set when *condor_run* is executed will be included in the environment of the Condor job.

condor_run removes the Condor job from the queue and deletes its temporary files, if *condor_run* is killed before the Condor job completes.

Options

-u *universe* Submit the job under the specified universe. The default is vanilla. While any universe may be specified, only the vanilla, standard, scheduler, and local universes result in a submit description file that may work properly.

Examples

condor_run may be used to compile an executable on a different platform. As an example, first set the environment variables for the required platform:

```
% setenv CONDOR_ARCH "SUN4u"
% setenv CONDOR_OPSYS "SOLARIS28"
```

Then, use *condor_run* to submit the compilation as in the following three examples.

```
% condor_run "f77 -O -o myprog myprog.f"
```

or

```
% condor_run "make"
```

or

```
% condor_run "condor_compile cc -o myprog.condor myprog.c"
```

Files

condor_run creates the following temporary files in the user's working directory. The placeholder <pid> is replaced by the process id of *condor_run*.

.condor_run.<pid> A shell script containing the shell command line.

.condor_submit.<pid> The submit description file for the job.

.condor_log.<pid> The Condor job's log file; it is monitored by *condor_run*, to determine when the job exits.

.condor_out.<pid> The output of the Condor job before it is output to the terminal.

.condor_error.<pid> Any error messages for the Condor job before they are output to the terminal.

condor_run removes these files when the job completes. However, if *condor_run* fails, it is possible that these files will remain in the user's working directory, and the Condor job may remain in the queue.

General Remarks

condor_run is intended for submitting simple shell command lines to Condor. It does not provide the full functionality of *condor_submit*. Therefore, some *condor_submit* errors and system failures may not be handled correctly.

All processes specified within the single shell command line will be executed on the single machine matched with the job. Condor will not distribute multiple processes of a command line pipe across multiple machines.

condor_run will use the shell specified in the `SHELL` environment variable, if one exists. Otherwise, it will use */bin/sh* to execute the shell command-line.

By default, *condor_run* expects Perl to be installed in `/usr/bin/perl`. If Perl is installed in another path, ask the Condor administrator to edit the path in the *condor_run* script, or explicitly call Perl from the command line:

```
% perl path-to-condor/bin/condor_run "shell-cmd"
```

Exit Status

condor_run exits with a status value of 0 (zero) upon complete success. The exit status of *condor_run* will be non-zero upon failure. The exit status in the case of a single error due to a system call will be the error number (`errno`) of the failed call.

Author

Condor Team, University of Wisconsin–Madison

Copyright

condor_set_shutdown

Set a program to execute upon *condor_master* shut down

Synopsis

condor_set_shutdown [**-help** | **-version**]

condor_set_shutdown **-exec** *programname* [**-debug**] [**-pool** *centralmanagerhostname[:portnumber]*] [**-name** *hostname* | *hostname* | **-addr** *"<a.b.c.d:port>"* | *"<a.b.c.d:port>"* | **-constraint** *expression* | **-all**]

Description

condor_set_shutdown sets a program (typically a script) to execute when the *condor_master* daemon shuts down. The **-exec** *programname* argument is required, and specifies the program to run. The string *programname* must match the string that defines `Name` in the configuration variable `MASTER_SHUTDOWN_<Name>` in the *condor_master* daemon's configuration. If it does not match, the *condor_master* will log an error and ignore the request.

For purposes of authentication and authorization, this command requires the `ADMINISTRATOR` access level. See section 3.6.1 on page 315 for further explanation.

Options

-help Display usage information

-version Display version information

-debug Causes debugging information to be sent to `stderr`, based on the value of the configuration variable `TOOL_DEBUG`

-pool *centralmanagerhostname[:portnumber]* Specify a pool by giving the central manager's host name and an optional port number

-name *hostname* Send the command to a machine identified by *hostname*

hostname Send the command to a machine identified by *hostname*

-addr *"<**a.b.c.d:port**>"* Send the command to a machine's master located at *"<a.b.c.d:port>"*

*"<**a.b.c.d:port**>"* Send the command to a machine located at *"<a.b.c.d:port>"*

-constraint ***expression*** Apply this command only to machines matching the given ClassAd *expression*

-all Send the command to all machines in the pool

Exit Status

condor_set_shutdown will exit with a status value of 0 (zero) upon success, and it will exit with the value 1 (one) upon failure.

Examples

To have all *condor_master* daemons run the program */bin/reboot* upon shut down, configure the *condor_master* to contain a definition similar to:

```
MASTER_SHUTDOWN_REBOOT = /sbin/reboot
```

where `REBOOT` is an invented name for this program that the *condor_master* will execute. On the command line, run

```
% condor_set_shutdown -exec reboot -all
% condor_off -graceful -all
```

where the string `reboot` matches the invented name.

Author

Condor Team, University of Wisconsin–Madison

Copyright

condor_ssh_to_job

create an ssh session to a running job

Synopsis

condor_ssh_to_job [**-help**]

condor_ssh_to_job [**-debug**] [**-name** *schedd-name*] [**-pool** *pool-name*] [**-ssh** *ssh-command*] [**-keygen-options** *ssh-keygen-options*] [**-shells** *shell1,shell2,...*] [**-auto-retry**] *cluster* | *cluster.process* | *cluster.process.node* [*remote-command*]

Description

condor_ssh_to_job creates an *ssh* session to a running job. The job is specified with the argument. If only the job *cluster* id is given, then the job *process* id defaults to the value 0.

It is available in Unix Condor distributions, and it works for vanilla, java, local, and parallel universe jobs. The user must be the owner of the job or must be a queue super user, and both the *condor_schedd* and *condor_starter* daemons must allow *condor_ssh_to_job* access. If no *remote-command* is specified, an interactive shell is created. An alternate *ssh* program such as *sftp* may be specified, using the **-ssh** option for uploading and downloading files.

The remote command or shell runs with the same user id as the running job, and it is initialized with the same working directory. The environment is initialized to be the same as that of the job, plus any changes made by the shell setup scripts and any environment variables passed by the *ssh* client. In addition, the environment variable `_CONDOR_JOB_PIDS` is defined. It is a space-separated list of PIDs associated with the job. At a minimum, the list will contain the PID of the process started when the job was launched, and it will be the first item in the list. It may contain additional PIDs of other processes that the job has created.

The *ssh* session and all processes it creates are treated by Condor as though they are processes belonging to the job. If the slot is preempted or suspended, the *ssh* session is killed or suspended along with the job. If the job exits before the *ssh* session finishes, the slot remains in the Claimed Busy state and is treated as though not all job processes have exited until all *ssh* sessions are closed. Multiple *ssh* sessions may be created to the same job at the same time. Resource consumption of the *sshd* process and all processes spawned by it are monitored by the *condor_starter* as though these processes belong to the job, so any policies such as `PREEMPT` that enforce a limit on resource consumption also take into account resources consumed by the *ssh* session.

condor_ssh_to_job stores ssh keys in temporary files within a newly created and uniquely named directory. The newly created directory will be within the directory defined by the environment variable `TMPDIR`. When the ssh session is finished, this directory and the ssh keys contained within it are removed.

See section 3.3.34 for details of the configuration variables related to *condor_ssh_to_job*.

An *ssh* session works by first authenticating and authorizing a secure connection between *condor_ssh_to_job* and the *condor_starter* daemon, using Condor protocols. The *condor_starter* generates an ssh key pair and sends it securely to *condor_ssh_to_job*. Then the *condor_starter* spawns *sshd* in inetd mode with its stdin and stdout attached to the TCP connection from *condor_ssh_to_job*. *condor_ssh_to_job* acts as a proxy for the *ssh* client to communicate with *sshd*, using the existing connection authorized by Condor. *At no point is sshd listening on the network for connections or running with any privileges other than that of the user identity running the job.* If CCB is being used to enable connectivity to the execute node from outside of a firewall or private network, *condor_ssh_to_job* is able to make use of CCB in order to form the *ssh* connection.

The login shell of the user id running the job is used to run the requested command, *sshd* subsystem, or interactive shell. This is hard-coded behavior in *OpenSSH* and cannot be overridden by configuration. This means that *condor_ssh_to_job* access is effectively disabled if the login shell disables access, as in the example programs */bin/true* and */sbin/nologin*.

condor_ssh_to_job is intended to work with *OpenSSH* as installed in typical environments. It does not work on Windows platforms. If the *ssh* programs are installed in non-standard locations, then the paths to these programs will need to be customized within the Condor configuration. Versions of *ssh* other than *OpenSSH* may work, but they will likely require additional configuration of command-line arguments, changes to the *sshd* configuration template file, and possibly modification of the `$(LIBEXEC)/condor_ssh_to_job_sshd_setup` script used by the *condor_starter* to set up *sshd*.

Options

-help Display brief usage information and exit.

-debug Causes debugging information to be sent to `stderr`, based on the value of the configuration variable `TOOL_DEBUG`

-name *schedd-name* Specify an alternate *condor_schedd*, if the default (local) one is not desired.

-pool *pool-name* Specify an alternate Condor pool, if the default one is not desired.

-ssh *ssh-command* Specify an alternate *ssh* program to run in place of *ssh*, for example *sftp* or *scp*. Additional arguments are specified as *ssh-command*. Since the arguments are delimited by spaces, place double quote marks around the whole command, to prevent the shell from splitting it into multiple arguments to *condor_ssh_to_job*. If any arguments must contain spaces, enclose them within single quotes.

-keygen-options *ssh-keygen-options* Specify additional arguments to the *ssh_keygen* program, for creating the ssh key that is used for the duration of the session. For example, a different number of bits could be used, or a different key type than the default.

-shells *shell1,shell2,...* Specify a comma-separated list of shells to attempt to launch. If the first shell does not exist on the remote machine, then the following ones in the list will be tried. If none of the specified shells can be found, */bin/sh* is used by default. If this option is not specified, it defaults to the environment variable SHELL from within the *condor_ssh_to_job* environment.

-auto-retry Specifies that if the job is not yet running, *condor_ssh_to_job* should keep trying periodically until it succeeds or encounters some other error.

Examples

```
% condor_ssh_to_job 32.0
Welcome to slot2@tonic.cs.wisc.edu!
Your condor job is running with pid(s) 65881.
% gdb -p 65881
(gdb) where
...
% logout
Connection to condor-job.tonic.cs.wisc.edu closed.
```

To upload or download files interactively with *sftp*:

```
% condor_ssh_to_job -ssh sftp 32.0
Connecting to condor-job.tonic.cs.wisc.edu...
sftp> ls
...
sftp> get outputfile.dat
```

This example shows downloading a file from the job with *scp*. The string "remote" is used in place of a host name in this example. It is not necessary to insert the correct remote host name, or even a valid one, because the connection to the job is created automatically. Therefore, the placeholder string "remote" is perfectly fine.

```
% condor_ssh_to_job -ssh scp 32 remote:outputfile.dat .
```

This example uses *condor_ssh_to_job* to accomplish the task of running *rsync* to synchronize a local file with a remote file in the job's working directory. Job id 32.0 is used in place of a host name in this example. This causes *rsync* to insert the expected job id in the arguments to *condor_ssh_to_job*.

```
% rsync -v -e "condor_ssh_to_job" 32.0:outputfile.dat .
```

Exit Status

condor_ssh_to_job will exit with a non-zero status value if it fails to set up an ssh session. If it succeeds, it will exit with the status value of the remote command or shell.

Author

Condor Team, University of Wisconsin–Madison

Copyright

condor_stats

Display historical information about the Condor pool

Synopsis

condor_stats [**-f** *filename*] [**-orgformat**] [**-pool** *centralmanagerhostname[:portnumber]*] [**time-range**] *query-type*

Description

condor_stats displays historic information about a Condor pool. Based on the type of information requested, a query is sent to the *condor_collector* daemon, and the information received is displayed using the standard output. If the **-f** option is used, the information will be written to a file instead of to standard output. The **-pool** option can be used to get information from other pools, instead of from the local (default) pool. The *condor_stats* tool is used to query resource information (single or by platform), submitter and user information, and checkpoint server information. If a time range is not specified, the default query provides information for the previous 24 hours. Otherwise, information can be retrieved for other time ranges such as the last specified number of hours, last week, last month, or a specified date range.

The information is displayed in columns separated by tabs. The first column always represents the time, as a percentage of the range of the query. Thus the first entry will have a value close to 0.0, while the last will be close to 100.0. If the **-orgformat** option is used, the time is displayed as number of seconds since the Unix epoch. The information in the remainder of the columns depends on the query type.

Note that logging of pool history must be enabled in the *condor_collector* daemon, otherwise no information will be available.

One query type is required. If multiple queries are specified, only the last one takes effect.

Time Range Options

-lastday Get information for the last day.

-lastweek Get information for the last week.

-lastmonth Get information for the last month.

-lasthours *n* Get information for the n last hours.

-from *m d y* Get information for the time since the beginning of the specified date. A start date prior to the Unix epoch causes *condor_stats* to print its usage information and quit.

-to *m d y* Get information for the time up to the beginning of the specified date, instead of up to now. A finish date in the future causes *condor_stats* to print its usage information and quit.

Query Type Arguments

The query types that do not list all of a category require further specification as given by an argument.

-resourcequery *hostname* A single resource query provides information about a single machine. The information also includes the keyboard idle time (in seconds), the load average, and the machine state.

-resourcelist A query of a single list of resources to provide a list of all the machines for which the *condor_collector* daemon has historic information within the query's time range.

-resgroupquery *arch/opsys* | *"Total"* A query of a specified group to provide information about a group of machines based on their platform (operating system and architecture). The architecture is defined by the machine ClassAd `Arch`, and the operating system is defined by the machine ClassAd `OpSys`. The string "Total" ask for information about all platforms.

The columns displayed are the number of machines that are unclaimed, matched, claimed, preempting, and in the owner state.

-resgrouplist Queries for a list of all the group names for which the *condor_collector* has historic information within the query's time range.

-userquery *email_address/submit_machine* Query for a specific submitter on a specific machine. The information displayed includes the number of running jobs and the number of idle jobs. An example argument appears as

```
-userquery jondoe@sample.com/onemachine.sample.com
```

-userlist Queries for the list of all submitters for which the *condor_collector* daemon has historic information within the query's time range.

-usergroupquery *email_address | "Total"* Query for all jobs submitted by the specific user, regardless of the machine they were submitted from, or all jobs. The information displayed includes the number of running jobs and the number of idle jobs.

-usergrouplist Queries for the list of all users for which the *condor_collector* has historic information within the query's time range.

-ckptquery *hostname* Query about a checkpoint server given its host name. The information displayed includes the number of Mbytes received, Mbytes sent, average receive bandwidth (in Kbytes/sec), and average send bandwidth (in Kbytes/sec).

-ckptlist Query for the entire list of checkpoint servers for which the *condor_collector* has historic information in the query's time range.

Options

-f *filename* Write the information to a file instead of the standard output.

-pool *centralmanagerhostname[:portnumber]* Contact the specified central manager instead of the local one.

-orgformat Display the information in an alternate format for timing, which presents timestamps since the Unix epoch. This argument only affects the display of *resoursequery*, *resgroupquery*, *userquery*, *usergroupquery*, and *ckptquery*.

Exit Status

condor_stats will exit with a status value of 0 (zero) upon success, and it will exit with the value 1 (one) upon failure.

Author

Condor Team, University of Wisconsin–Madison

Copyright

condor_status

Display status of the Condor pool

Synopsis

condor_status [**-debug**] [*help options*] [*query options*] [*display options*] [*custom options*] [*name . . .*]

Description

condor_status is a versatile tool that may be used to monitor and query the Condor pool. The *condor_status* tool can be used to query resource information, submitter information, checkpoint server information, and daemon master information. The specific query sent and the resulting information display is controlled by the query options supplied. Queries and display formats can also be customized.

The options that may be supplied to *condor_status* belong to five groups:

- **Help options** provide information about the *condor_status* tool.
- **Query options** control the content and presentation of status information.
- **Display options** control the display of the queried information.
- **Custom options** allow the user to customize query and display information.
- **Host options** specify specific machines to be queried

At any time, only one *help option,* one *query option* and one *custom option* may be specified. Any number of *custom* and *host options* may be specified.

Options

-debug Causes debugging information to be sent to `stderr`, based on the value of the configuration variable `TOOL_DEBUG`

-help (Help option) Display usage information

-diagnose (Help option) Print out query ClassAd without performing query

-any (Query option) Query all ClassAds and display their type, target type, and name

-avail (Query option) Query *condor_startd* ClassAds and identify resources which are available

-ckptsrvr (Query option) Query *condor_ckpt_server* ClassAds and display checkpoint server attributes

-claimed (Query option) Query *condor_startd* ClassAds and print information about claimed resources

-cod (Query option) Display only machine ClassAds that have COD claims. Information displayed includes the claim ID, the owner of the claim, and the state of the COD claim.

-collector (Query option) Query *condor_collector* ClassAds and display attributes

-direct *hostname* (Query option) Go directly to the given host name to get the ClassAds to display

-java (Query option) Display only Java-capable resources.

-license (Query option) Display license attributes.

-master (Query option) Query *condor_master* ClassAds and display daemon master attributes

-negotiator (Query option) Query *condor_negotiator* ClassAds and display attributes

-pool *centralmanagerhostname[:portnumber]* (Query option) Query the specified central manager using an optional port number. *condor_status* queries the machine specified by the configuration variable `COLLECTOR_HOST` by default.

-quill (Query option) Display attributes of machines running Quill.

-run (Query option) Display information about machines currently running jobs.

-schedd (Query option) Query *condor_schedd* ClassAds and display attributes

-server (Query option) Query *condor_startd* ClassAds and display resource attributes

-startd (Query option) Query *condor_startd* ClassAds

-state (Query option) Query *condor_startd* ClassAds and display resource state information

-storage (Query option) Display attributes of machines with network storage resources.

-submitters (Query option) Query ClassAds sent by submitters and display important submitter attributes

-subsystem *type* (Query option) If *type* is one of *collector*, *negotiator*, *master*, *schedd*, *startd*, or *quill*, then behavior is the same as the query option without the **-subsystem** option. For example, **-subsystem** *collector* is the same as **-collector**. A value of *type* of *CkptServer*, *Machine*, *DaemonMaster*, or *Scheduler* targets that type of ClassAd.

-vm (Query option) Query *condor_startd* ClassAds, and display only VM-enabled machines. Information displayed includes the the machine name, the virtual machine software version, the state of machine, the virtual machine memory, and the type of networking.

-attributes *Attr1 [,Attr2* . . .*]* (Display option) Explicitly list the attributes in a comma separated list which should be displayed when using the **-xml** or **-long** options. Limiting the number of attributes increases the efficiency of the query.

-expert (Display option) Display shortened error messages

-long (Display option) Display entire ClassAds (same as **-verbose**)

-sort *attr* (Display option) Display entries in ascending order based on the value of the named attribute

-total (Display option) Display totals only

-verbose (Display option) Display entire ClassAds. Implies that totals will not be displayed.

-xml (Display option) Display entire ClassAds, in XML format. The XML format is fully defined at http://www.cs.wisc.edu/condor/classad/refman/.

-constraint *const* (Custom option) Add constraint expression. See section **??** for details on writing expressions.

-format *fmt attr* (Custom option) Display attribute or expression *attr* in format *fmt*. To display the attribute or expression the format must contain a single `printf(3)` style conversion specifier. Attributes must be from the resource ClassAd. Expressions are ClassAd expressions and may refer to attributes in the resource ClassAd. If the attribute is not present in a given ClassAd and cannot be parsed as an expression, then the format option will be silently skipped. The conversion specifier must match the type of the attribute or expression. %s is suitable for strings such as `Name`, %d for integers such as `LastHeardFrom`, and %f for floating point numbers such as `LoadAvg`. An incorrect format will result in undefined behavior. Do not use more than one conversion specifier in a given format. More than one conversion specifier will result in undefined behavior. To output multiple attributes repeat the **-format** option once for each desired attribute. Like `printf(3)` style formats, one may include other text that will be reproduced directly. A format without any conversion specifiers may be specified, but an attribute is still required. Include \n to specify a line break.

General Remarks

- The default output from *condor_status* is formatted to be human readable, not script readable. In an effort to make the output fit within 80 characters, values in some fields might be truncated. Furthermore, the Condor Project can (and does) change the formatting of this default output as we see fit. Therefore, any script that is attempting to parse data from *condor_status* is strongly encouraged to use the **-format** option (described above).
- The information obtained from *condor_startd* and *condor_schedd* daemons may sometimes appear to be inconsistent. This is normal since *condor_startd* and *condor_schedd* daemons update the Condor manager at different rates, and since there is a delay as information propagates through the network and the system.
- Note that the `ActivityTime` in the `Idle` state is *not* the amount of time that the machine has been idle. See the section on *condor_startd* states in the *Administrator's Manual* for more information.
- When using *condor_status* on a pool with SMP machines, you can either provide the host name, in which case you will get back information about all slots that are represented on that host, or you can list specific slots by name. See the examples below for details.
- If you specify host names, without domains, Condor will automatically try to resolve those host names into fully qualified host names for you. This also works when specifying specific nodes of an SMP machine. In this case, everything after the "@" sign is treated as a host name and that is what is resolved.
- You can use the **-direct** option in conjunction with almost any other set of options. However, at this time, the only daemon that will allow direct queries for its ad(s) is the *condor_startd*.

So, the only options currently not supported with **-direct** are **-schedd** and **-master**. Most other options use startd ads for their information, so they work seamlessly with **-direct**. The only other restriction on **-direct** is that you may only use 1 **-direct** option at a time. If you want to query information directly from multiple hosts, you must run *condor_status* multiple times.

- Unless you use the local host name with **-direct**, *condor_status* will still have to contact a collector to find the address where the specified daemon is listening. So, using a **-pool** option in conjunction with **-direct** just tells *condor_status* which collector to query to find the address of the daemon you want. The information actually displayed will still be retrieved directly from the daemon you specified as the argument to **-direct**.

Examples

Example 1 To view information from all nodes of an SMP machine, use only the host name. For example, if you had a 4-CPU machine, named `vulture.cs.wisc.edu`, you might see

```
% condor_status vulture

Name               OpSys      Arch   State     Activity LoadAv Mem   ActvtyTime

slot1@vulture.cs.w LINUX      INTEL  Claimed   Busy     1.050   512  0+01:47:42
slot2@vulture.cs.w LINUX      INTEL  Claimed   Busy     1.000   512  0+01:48:19
slot3@vulture.cs.w LINUX      INTEL  Unclaimed Idle     0.070   512  1+11:05:32
slot4@vulture.cs.w LINUX      INTEL  Unclaimed Idle     0.000   512  1+11:05:34

                     Total Owner Claimed Unclaimed Matched Preempting Backfill

         INTEL/LINUX     4     0       2         2       0          0        0

               Total     4     0       2         2       0          0        0
```

Example 2 To view information from a specific nodes of an SMP machine, specify the node directly. You do this by providing the name of the slot. This has the form `slot#@hostname`. For example:

```
% condor_status slot3@vulture

Name               OpSys      Arch   State     Activity LoadAv Mem   ActvtyTime

slot3@vulture.cs.w LINUX      INTEL  Unclaimed Idle     0.070   512  1+11:10:32

                     Total Owner Claimed Unclaimed Matched Preempting Backfill

         INTEL/LINUX     1     0       0         1       0          0        0

               Total     1     0       0         1       0          0        0
```

Constraint option examples

Further explanation and examples are in section 4.1.4.

The Unix command to use the constraint option to see all machines with the `OpSys` of `"LINUX"`:

```
% condor_status -constraint OpSys==\"LINUX\"
```

Note that quotation marks must be escaped with the backslash characters for most shells.

The Windows command to do the same thing:

```
>condor_status -constraint " OpSys==""LINUX"" "
```

Note that quotation marks are used to delimit the single argument which is the expression, and the quotation marks that identify the string must be escaped by using a set of two double quote marks without any intervening spaces.

To see all machines that are currently in the Idle state, the Unix command is

```
% condor_status -constraint State==\"Idle\"
```

To see all machines that are bench marked to have a MIPS rating of more than 750, the Unix command is

```
% condor_status -constraint 'Mips>750'
```

-cod option example

The **-cod** option displays the status of COD claims within a given Condor pool.

```
Name        ID   ClaimState TimeInState RemoteUser JobId Keyword
astro.cs.wi COD1 Idle        0+00:00:04 wright
chopin.cs.w COD1 Running     0+00:02:05 wright     3.0   fractgen
chopin.cs.w COD2 Suspended   0+00:10:21 wright     4.0   fractgen

               Total  Idle  Running  Suspended  Vacating  Killing
 INTEL/LINUX       3     1        1          1         0        0
       Total       3     1        1          1         0        0
```

Exit Status

condor_status will exit with a status value of 0 (zero) upon success, and it will exit with the value 1 (one) upon failure.

Author

Condor Team, University of Wisconsin–Madison

Copyright

condor_store_cred

securely stash a password

Synopsis

condor_store_cred [**-help**]

condor_store_cred *add* [**-c | -u** *username*][**-p** *password*] [**-n** *machinename*] [**-f** *filename*]

condor_store_cred *delete* [**-c | -u** *username*][**-n** *machinename*]

condor_store_cred *query* [**-c | -u** *username*][**-n** *machinename*]

Description

condor_store_cred stores passwords in a secure manner. There are two separate uses of *condor_store_cred*:

1. A shared pool password is needed in order to implement the `PASSWORD` authentication method. *condor_store_cred* using the **-c** option deals with the password for the implied `condor_pool@$(UID_DOMAIN)` user name.

   On a Unix machine, *condor_store_cred* with the **-f** option is used to set the pool password, as needed when used with the `PASSWORD` authentication method. The pool password is placed in a file specified by the `SEC_PASSWORD_FILE` configuration variable.

2. In order to submit a job from a Windows platform machine, or to execute a job on a Windows platform machine utilizing the **run_as_owner** functionality, *condor_store_cred* stores the password of a user/domain pair securely in the Windows registry. Using this stored password, Condor may act on behalf of the submitting user to access files, such as writing output or log files. Condor is able to run jobs with the user ID of the submitting user. The password is stored in the same manner as the system does when setting or changing account passwords.

Passwords are stashed in a persistent manner; they are maintained across system reboots.

The *add* argument on the Windows platform stores the password securely in the registry. The user is prompted to enter the password twice for confirmation, and characters are not echoed. If there is already a password stashed, the old password will be overwritten by the new password.

The *delete* argument deletes the current password, if it exists.

The *query* reports whether the password is stored or not.

Options

-c Operations refer to the pool password, as used in the PASSWORD authentication method.

-f *filename* For Unix machines only, generates a pool password file named *filename* that may be used with the PASSWORD authentication method.

-help Displays a brief summary of command options.

-n *machinename* Apply the command on the given machine.

-p *password* Stores *password*, rather than prompting the user to enter a password.

-u *username* Specify the user name.

Exit Status

condor_store_cred will exit with a status value of 0 (zero) upon success, and it will exit with the value 1 (one) upon failure.

Author

Condor Team, University of Wisconsin–Madison

Copyright

condor_submit

Queue jobs for execution under Condor

Synopsis

condor_submit [**-verbose**] [**-unused**] [**-name** *schedd_name*] [**-remote** *schedd_name*] [**-pool** *pool_name*] [**-disable**] [**-password** *passphrase*] [**-debug**] [**-append** *command . . .*][**-spool**] [**-dump** *filename*] [*submit description file*]

Description

condor_submit is the program for submitting jobs for execution under Condor. *condor_submit* requires a submit description file which contains commands to direct the queuing of jobs. One submit description file may contain specifications for the queuing of many Condor jobs at once. A single invocation of *condor_submit* may cause one or more clusters. A cluster is a set of jobs specified in the submit description file between **queue** commands for which the executable is not changed. It is advantageous to submit multiple jobs as a single cluster because:

- Only one copy of the checkpoint file is needed to represent all jobs in a cluster until they begin execution.
- There is much less overhead involved for Condor to start the next job in a cluster than for Condor to start a new cluster. This can make a big difference when submitting lots of short jobs.

Multiple clusters may be specified within a single submit description file. Each cluster must specify a single executable.

The job ClassAd attribute `ClusterId` identifies a cluster. See specifics for this attribute in the Appendix on page 902.

Note that submission of jobs from a Windows machine requires a stashed password to allow Condor to impersonate the user submitting the job. To stash a password, use the *condor_store_cred* command. See the manual page at page 823 for details.

For lengthy lines within the submit description file, the backslash (\) is a line continuation character. Placing the backslash at the end of a line causes the current line's command to be continued with the next line of the file. Submit description files may contain comments. A comment is any line beginning with a pound character (#).

Here is a list of the commands that may be placed in the submit description file to direct the submission of a job.

Options

-verbose Verbose output - display the created job ClassAd

-unused As a default, causes no warnings to be issued about user-defined macros not being used within the submit description file. The meaning reverses (toggles) when the configuration variable `WARN_ON_UNUSED_SUBMIT_FILE_MACROS` is set to the nondefault value of `False`. Printing the warnings can help identify spelling errors of submit description file commands. The warnings are sent to stderr.

-name *schedd_name* Submit to the specified *condor_schedd*. Use this option to submit to a *condor_schedd* other than the default local one. *schedd_name* is the value of the `Name` ClassAd attribute on the machine where the *condor_schedd* daemon runs.

-remote *schedd_name* Submit to the specified *condor_schedd*, spooling all required input files over the network connection. *schedd_name* is the value of the `Name` ClassAd attribute on the machine where the *condor_schedd* daemon runs. This option is equivalent to using both **-name** and **-spool**.

-pool *pool_name* Look in the specified pool for the *condor_schedd* to submit to. This option is used with **-name** or **-remote**.

-disable Disable file permission checks.

-password *passphrase* Specify a password to the *MyProxy* server.

-debug Cause debugging information to be sent to `stderr`, based on the value of the configuration variable `TOOL_DEBUG`.

-append *command* Augment the commands in the submit description file with the given command. This command will be considered to immediately precede the Queue command within the submit description file, and come after all other previous commands. The submit description file is not modified. Multiple commands are specified by using the **-append** option multiple times. Each new command is given in a separate **-append** option. Commands with spaces in them will need to be enclosed in double quote marks.

-spool Spool all required input files, user log, and proxy over the connection to the *condor_schedd*. After submission, modify local copies of the files without affecting your jobs. Any output files for completed jobs need to be retrieved with *condor_transfer_data*.

-dump *filename* Sends all ClassAds to the specified file, instead of to the *condor_schedd*.

submit description file The pathname to the submit description file. If this optional argument is missing or equal to "-", then the commands are taken from standard input.

Submit Description File Commands

Each submit description file describes one cluster of jobs to be placed in the Condor execution pool. All jobs in a cluster must share the same executable, but they may have different input and output files, and different program arguments. The submit description file is the only command-line argument to *condor_submit*. If the submit description file argument is omitted, *condor_submit* will read the submit description from standard input.

The submit description file must contain one *executable* command and at least one *queue* command. All of the other commands have default actions.

The commands which can appear in the submit description file are numerous. They are listed here in alphabetical order by category.

BASIC COMMANDS

arguments = <**argument_list**> List of arguments to be supplied to the executable as part of the command line.

In the **java** universe, the first argument must be the name of the class containing `main`.

There are two permissible formats for specifying arguments, identified as the old syntax and the new syntax. The old syntax supports white space characters within arguments only in special circumstances, hence the new syntax, which supports uniform quoting of white space characters within arguments.

Old Syntax

In the old syntax, individual command line arguments are delimited (separated) by space characters. To allow a double quote mark in an argument, it is escaped with a backslash; that is, the two character sequence `\"` becomes a single double quote mark within an argument.

Further interpretation of the argument string differs depending on the operating system. On Windows, the entire argument string is passed verbatim (other than the backslash in front of double quote marks) to the Windows application. Most Windows applications will allow spaces within an argument value by surrounding the argument with double quotes marks. In all other cases, there is no further interpretation of the arguments.

Example:

```
arguments = one \"two\" 'three'
```

Produces in Unix vanilla universe:

```
argument 1: one
argument 2: "two"
argument 3: 'three'
```

New Syntax

Here are the rules for using the new syntax:

1. The entire string representing the command line arguments is surrounded by double quote marks. This permits the white space characters of spaces and tabs to potentially be embedded within a single argument. Putting the double quote mark within the arguments is accomplished by escaping it with another double quote mark.
2. The white space characters of spaces or tabs delimit arguments.
3. To embed white space characters of spaces or tabs within a single argument, surround the entire argument with single quote marks.
4. To insert a literal single quote mark, escape it within an argument already delimited by single quote marks by adding another single quote mark.

Example:

```
arguments = "3 simple arguments"
```

Produces:

```
argument 1: 3
argument 2: simple
argument 3: arguments
```

Another example:

```
arguments = "one 'two with spaces' 3"
```

Produces:

```
argument 1: one
argument 2: two with spaces
argument 3: 3
```

And yet another example:

```
arguments = "one ""two"" 'spacey ''quoted'' argument'"
```

Produces:

```
argument 1: one
argument 2: "two"
argument 3: spacey 'quoted' argument
```

Notice that in the new syntax, the backslash has no special meaning. This is for the convenience of Windows users.

environment = <**parameter_list**> List of environment variables.

There are two different formats for specifying the environment variables: the old format and the new format. The old format is retained for backward-compatibility. It suffers from a platform-dependent syntax and the inability to insert some special characters into the environment.

The new syntax for specifying environment values:

1. Put double quote marks around the entire argument string. This distinguishes the new syntax from the old. The old syntax does not have double quote marks around it. Any literal double quote marks within the string must be escaped by repeating the double quote mark.
2. Each environment entry has the form

   ```
   <name>=<value>
   ```

3. Use white space (space or tab characters) to separate environment entries.
4. To put any white space in an environment entry, surround the space and as much of the surrounding entry as desired with single quote marks.
5. To insert a literal single quote mark, repeat the single quote mark anywhere inside of a section surrounded by single quote marks.

Example:

```
environment = "one=1 two=""2"" three='spacey ''quoted'' value'"
```

Produces the following environment entries:

```
one=1
two="2"
three=spacey 'quoted' value
```

Under the old syntax, there are no double quote marks surrounding the environment specification. Each environment entry remains of the form

```
<name>=<value>
```

Under Unix, list multiple environment entries by separating them with a semicolon (;). Under Windows, separate multiple entries with a vertical bar (|). There is no way to insert a literal semicolon under Unix or a literal vertical bar under Windows. Note that spaces are accepted, but rarely desired, characters within parameter names and values, because they are treated as literal characters, not separators or ignored white space. Place spaces within the parameter list only if required.

A Unix example:

```
environment = one=1;two=2;three="quotes have no 'special' meaning"
```

This produces the following:

```
one=1
two=2
three="quotes have no 'special' meaning"
```

If the environment is set with the **environment** command *and* **getenv** is also set to true, values specified with **environment** override values in the submittor's environment (regardless of the order of the **environment** and **getenv** commands).

error = <**pathname**> A path and file name used by Condor to capture any error messages the program would normally write to the screen (that is, this file becomes `stderr`). A path is given with respect to the file system of the machine on which the job is submitted. The file is written (by the job) in the remote scratch directory of the machine where the job is executed. When the job exits, the resulting file is transferred back to the machine where the job was submitted, and the path is utilized for file placement. If not specified, the default value of `/dev/null` is used for submission to a Unix machine. If not specified, error messages are ignored for submission to a Windows machine. More than one job should not use the same error file, since this will cause one job to overwrite the errors of another. The error file and the output file should not be the same file as the outputs will overwrite each other or be lost. For grid universe jobs, **error** may be a URL that the Globus tool *globus_url_copy* understands.

executable = <**pathname**> An optional path and a required file name of the executable file for this job cluster. Only one **executable** command within a submit description file is guaranteed to work properly. More than one often works.

If no path or a relative path is used, then the executable file is presumed to be relative to the current working directory of the user as the *condor_submit* command is issued.

If submitting into the standard universe, then the named executable must have been re-linked with the Condor libraries (such as via the *condor_compile* command). If submitting into the vanilla universe (the default), then the named executable need not be re-linked and can be any process which can run in the background (shell scripts work fine as well). If submitting into the Java universe, then the argument must be a compiled `.class` file.

getenv = <**True** | **False**> If **getenv** is set to `True`, then *condor_submit* will copy all of the user's current shell environment variables at the time of job submission into the job ClassAd. The job will therefore execute with the same set of environment variables that the user had at submit time. Defaults to `False`.

If the environment is set with the **environment** command *and* **getenv** is also set to true, values specified with **environment** override values in the submittor's environment (regardless of the order of the **environment** and **getenv** commands).

input = <**pathname**> Condor assumes that its jobs are long-running, and that the user will not wait at the terminal for their completion. Because of this, the standard files which normally access the terminal, (`stdin`, `stdout`, and `stderr`), must refer to files. Thus, the file name

specified with **input** should contain any keyboard input the program requires (that is, this file becomes `stdin`). A path is given with respect to the file system of the machine on which the job is submitted. The file is transferred before execution to the remote scratch directory of the machine where the job is executed. If not specified, the default value of `/dev/null` is used for submission to a Unix machine. If not specified, input is ignored for submission to a Windows machine. For grid universe jobs, **input** may be a URL that the Globus tool *globus_url_copy* understands.

Note that this command does *not* refer to the command-line arguments of the program. The command-line arguments are specified by the **arguments** command.

log = <**pathname**> Use **log** to specify a file name where Condor will write a log file of what is happening with this job cluster. For example, Condor will place a log entry into this file when and where the job begins running, when the job produces a checkpoint, or moves (migrates) to another machine, and when the job completes. Most users find specifying a **log** file to be handy; its use is recommended. If no **log** entry is specified, Condor does not create a log for this cluster.

log_xml = <**True** | **False**> If **log_xml** is `True`, then the log file will be written in ClassAd XML. If not specified, XML is not used. Note that the file is an XML fragment; it is missing the file header and footer. Do not mix XML and non-XML within a single file. If multiple jobs write to a single log file, ensure that all of the jobs specify this option in the same way.

notification = <**Always** | **Complete** | **Error** | **Never**> Owners of Condor jobs are notified by e-mail when certain events occur. If defined by *Always*, the owner will be notified whenever the job produces a checkpoint, as well as when the job completes. If defined by *Complete* (the default), the owner will be notified when the job terminates. If defined by *Error*, the owner will only be notified if the job terminates abnormally. If defined by *Never*, the owner will not receive e-mail, regardless to what happens to the job. The statistics included in the e-mail are documented in section 2.6.7 on page 49.

notify_user = <**email-address**> Used to specify the e-mail address to use when Condor sends e-mail about a job. If not specified, Condor defaults to using the e-mail address defined by

```
job-owner@UID_DOMAIN
```

where the configuration variable `UID_DOMAIN` is specified by the Condor site administrator. If `UID_DOMAIN` has not been specified, Condor sends the e-mail to:

```
job-owner@submit-machine-name
```

output = <**pathname**> The **output** file captures any information the program would ordinarily write to the screen (that is, this file becomes `stdout`). A path is given with respect to the file system of the machine on which the job is submitted. The file is written (by the job) in the remote scratch directory of the machine where the job is executed. When the job exits, the resulting file is transferred back to the machine where the job was submitted, and the path is utilized for file placement. If not specified, the default value of `/dev/null` is used for submission to a Unix machine. If not specified, output is ignored for submission to a Windows

machine. Multiple jobs should not use the same output file, since this will cause one job to overwrite the output of another. The output file and the error file should not be the same file as the outputs will overwrite each other or be lost. For grid universe jobs, **output** may be a URL that the Globus tool *globus_url_copy* understands.

Note that if a program explicitly opens and writes to a file, that file should *not* be specified as the **output** file.

priority = <**integer**> A Condor job priority can be any integer, with 0 being the default. Jobs with higher numerical priority will run before jobs with lower numerical priority. Note that this priority is on a per user basis. One user with many jobs may use this command to order his/her own jobs, and this will have no effect on whether or not these jobs will run ahead of another user's jobs.

queue [number-of-procs] Places one or more copies of the job into the Condor queue. The optional argument *number-of-procs* specifies how many times to submit the job to the queue, and it defaults to 1. If desired, any commands may be placed between subsequent **queue** commands, such as new **input**, **output**, **error**, **initialdir**, or **arguments** commands. This is handy when submitting multiple runs into one cluster with one submit description file.

universe = <**vanilla** | **standard** | **scheduler** | **local** | **grid** | **java** | **vm**> Specifies which Condor Universe to use when running this job. The Condor Universe specifies a Condor execution environment. The **standard** Universe tells Condor that this job has been re-linked via *condor_compile* with the Condor libraries and therefore supports checkpointing and remote system calls. The **vanilla** Universe is the default (except where the configuration variable `DEFAULT_UNIVERSE` defines it otherwise), and is an execution environment for jobs which have not been linked with the Condor libraries. *Note:* Use the **vanilla** Universe to submit shell scripts to Condor. The **scheduler** is for a job that should act as a metascheduler. The **grid** universe forwards the job to an external job management system. Further specification of the **grid** universe is done with the **grid_resource** command. The **java** universe is for programs written to the Java Virtual Machine. The **vm** universe facilitates the execution of a virtual machine.

COMMANDS FOR MATCHMAKING

rank = <**ClassAd Float Expression**> A ClassAd Floating-Point expression that states how to rank machines which have already met the requirements expression. Essentially, rank expresses preference. A higher numeric value equals better rank. Condor will give the job the machine with the highest rank. For example,

```
requirements = Memory > 60
rank = Memory
```

asks Condor to find all available machines with more than 60 megabytes of memory and give to the job the machine with the most amount of memory. See section 2.5.2 within the Condor Users Manual for complete information on the syntax and available attributes that can be used in the ClassAd expression.

requirements = <**ClassAd Boolean Expression**> The requirements command is a boolean ClassAd expression which uses C-like operators. In order for any job in this cluster to run on a given machine, this requirements expression must evaluate to true on the given machine. For example, to require that whatever machine executes a Condor job has a least 64 Meg of RAM and has a MIPS performance rating greater than 45, use:

```
requirements = Memory >= 64 && Mips > 45
```

For scheduler and local universe jobs, the requirements expression is evaluated against the `Scheduler` ClassAd which represents the the *condor_schedd* daemon running on the submit machine, rather than a remote machine. Like all commands in the submit description file, if multiple requirements commands are present, all but the last one are ignored. By default, *condor_submit* appends the following clauses to the requirements expression:

1. Arch and OpSys are set equal to the Arch and OpSys of the submit machine. In other words: unless you request otherwise, Condor will give your job machines with the same architecture and operating system version as the machine running *condor_submit*.
2. Disk >= DiskUsage. The `DiskUsage` attribute is initialized to the size of the executable plus the size of any files specified in a **transfer_input_files** command. It exists to ensure there is enough disk space on the target machine for Condor to copy over both the executable and needed input files. The `DiskUsage` attribute represents the maximum amount of total disk space required by the job in kilobytes. Condor automatically updates the `DiskUsage` attribute approximately every 20 minutes while the job runs with the amount of space being used by the job on the execute machine.
3. (Memory * 1024) >= ImageSize. To ensure the target machine has enough memory to run your job.
4. If Universe is set to Vanilla, FileSystemDomain is set equal to the submit machine's FileSystemDomain.

View the requirements of a job which has already been submitted (along with everything else about the job ClassAd) with the command *condor_q -l*; see the command reference for *condor_q* on page 772. Also, see the Condor Users Manual for complete information on the syntax and available attributes that can be used in the ClassAd expression.

FILE TRANSFER COMMANDS

should_transfer_files = <**YES** | **NO** | **IF_NEEDED** > The **should_transfer_files** setting is used to define if Condor should transfer files to and from the remote machine where the job runs. The file transfer mechanism is used to run jobs which are not in the standard universe (and can therefore use remote system calls for file access) on machines which do not have a shared file system with the submit machine. **should_transfer_files** equal to *YES* will cause Condor to always transfer files for the job. *NO* disables Condor's file transfer mechanism. *IF_NEEDED* will not transfer files for the job if it is matched with a resource in the same `FileSystemDomain` as the submit machine (and therefore, on a machine with the same shared file system). If the job is matched with a remote resource in a different `FileSystemDomain`, Condor will transfer the necessary files.

If defining **should_transfer_files** you *must* also define **when_to_transfer_output** (described below). For more information about this and other settings related to transferring files, see section 2.5.4 on page 25.

Note that **should_transfer_files** is not supported for jobs submitted to the grid universe.

stream_error = <**True** | **False**> If `True`, then `stderr` is streamed back to the machine from which the job was submitted. If `False`, `stderr` is stored locally and transferred back when the job completes. This command is ignored if the job ClassAd attribute `TransferErr` is `False`. The default value is `True` in the grid universe and `False` otherwise. This command must be used in conjunction with **error**, otherwise `stderr` will sent to `/dev/null` on Unix machines and ignored on Windows machines.

stream_input = <**True** | **False**> If `True`, then `stdin` is streamed from the machine on which the job was submitted. The default value is `False`. The command is only relevant for jobs submitted to the vanilla or java universes, and it is ignored by the grid universe. This command must be used in conjunction with **input**, otherwise `stdin` will be `/dev/null` on Unix machines and ignored on Windows machines.

stream_output = <**True** | **False**> If `True`, then `stdout` is streamed back to the machine from which the job was submitted. If `False`, `stdout` is stored locally and transferred back when the job completes. This command is ignored if the job ClassAd attribute `TransferOut` is `False`. The default value is `True` in the grid universe and `False` otherwise. This command must be used in conjunction with **output**, otherwise `stdout` will sent to `/dev/null` on Unix machines and ignored on Windows machines.

transfer_executable = <**True** | **False**> This command is applicable to jobs submitted to the grid and vanilla universes. If **transfer_executable** is set to `False`, then Condor looks for the executable on the remote machine, and does not transfer the executable over. This is useful for an already pre-staged executable; Condor behaves more like rsh. The default value is `True`.

transfer_input_files = < **file1,file2,file...** > A comma-delimited list of all the files and directories to be transferred into the working directory for the job, before the job is started. By default, the file specified in the **executable** command and any file specified in the **input** command (for example, `stdin`) are transferred.

When a path to an input file or directory is specified, this specifies the path to the file on the submit side. The file is placed in the job's temporary scratch directory on the execute side, and it is named using the base name of the original path. For example, `/path/to/input_file` becomes `input_file` in the job's scratch directory.

A directory may be specified using a trailing path separator. An example of a trailing path separator is the slash character on Unix platforms; a directory example using a trailing path separator is `input_data/`. When a directory is specified with a trailing path separator, the contents of the directory are transferred, but the directory itself is not transferred. It is as if each of the items within the directory were listed in the transfer list. When there is no trailing path separator, the directory is transferred, its contents are transferred, and these contents are placed inside the transferred directory.

For grid universe jobs other than Condor-C, the transfer of directories is not currently supported.

Symbolic links to files are transferred as the files they point to. Transfer of symbolic links to directories is not currently supported.

For vanilla and vm universe jobs only, a file may be specified by giving a URL, instead of a file name. The implementation for URL transfers requires both configuration and available plug-in. See section 3.13.3 for details.

For more information about this and other settings related to transferring files, see section 2.5.4 on page 25.

transfer_output_files = < **file1,file2,file...** > This command forms an explicit list of output files and directories to be transferred back from the temporary working directory on the execute machine to the submit machine. If there are multiple files, they must be delimited with commas.

For Condor-C jobs and all other non-grid universe jobs, if **transfer_output_files** is not specified, Condor will automatically transfer back all files in the job's temporary working directory which have been modified or created by the job. Subdirectories are not scanned for output, so if output from subdirectories is desired, the output list must be explicitly specified. For grid universe jobs other than Condor-C, desired output files must also be explicitly listed. Another reason to explicitly list output files is for a job that creates many files, and the user wants only a subset transferred back.

For grid universe jobs other than with grid type **condor**, to have files other than standard output and standard error transferred from the execute machine back to the submit machine, do use **transfer_output_files**, listing all files to be transferred. These files are found on the execute machine in the working directory of the job.

When a path to an output file or directory is specified, it specifies the path to the file on the execute side. As a destination on the submit side, the file is placed in the job's initial working directory, and it is named using the base name of the original path. For example, `path/to/output_file` becomes `output_file` in the job's initial working directory. The name and path of the file that is written on the submit side may be modified by using **transfer_output_remaps**.

A directory may be specified using a trailing path separator. An example of a trailing path separator is the slash character on Unix platforms; a directory example using a trailing path separator is `input_data/`. When a directory is specified with a trailing path separator, the contents of the directory are transferred, but the directory itself is not transferred. It is as if each of the items within the directory were listed in the transfer list. When there is no trailing path separator, the directory is transferred, its contents are transferred, and these contents are placed inside the transferred directory.

For grid universe jobs other than Condor-C, the transfer of directories is not currently supported.

Symbolic links to files are transferred as the files they point to. Transfer of symbolic links to directories is not currently supported.

For more information about this and other settings related to transferring files, see section 2.5.4 on page 25.

transfer_output_remaps = < **" name = newname ; name2 = newname2 ... "**> This specifies the name (and optionally path) to use when downloading output files from the completed job. Normally, output files are transferred back to the initial working directory with the same name they had in the execution directory. This gives you the option to save them with a different path or name. If you specify a relative path, the final path will be relative to the job's initial working directory.

name describes an output file name produced by your job, and *newname* describes the file name it should be downloaded to. Multiple remaps can be specified by separating each with a semicolon. If you wish to remap file names that contain equals signs or semicolons, these special characters may be escaped with a backslash.

when_to_transfer_output = < **ON_EXIT** | **ON_EXIT_OR_EVICT** > Setting **when_to_transfer_output** equal to *ON_EXIT* will cause Condor to transfer the job's output files back to the submitting machine only when the job completes (exits on its own).

The *ON_EXIT_OR_EVICT* option is intended for fault tolerant jobs which periodically save their own state and can restart where they left off. In this case, files are spooled to the submit machine any time the job leaves a remote site, either because it exited on its own, or was evicted by the Condor system for any reason prior to job completion. The files spooled back are placed in a directory defined by the value of the `SPOOL` configuration variable. Any output files transferred back to the submit machine are automatically sent back out again as input files if the job restarts.

For more information about this and other settings related to transferring files, see section 2.5.4 on page 25.

POLICY COMMANDS

hold = <**True** | **False**> If **hold** is set to `True`, then the submitted job will be placed into the Hold state. Jobs in the Hold state will not run until released by *condor_release*. Defaults to `False`.

leave_in_queue = <**ClassAd Boolean Expression**> When the ClassAd Expression evaluates to `True`, the job is not removed from the queue upon completion. This allows the user of a remotely spooled job to retrieve output files in cases where Condor would have removed them as part of the cleanup associated with completion. The job will only exit the queue once it has been marked for removal (via *condor_rm*, for example) and the **leave_in_queue** expression has become `False`. **leave_in_queue** defaults to `False`.

As an example, if the job is to be removed once the output is retrieved with *condor_transfer_data*, then use

```
leave_in_queue = (JobStatus == 4) && ((StageOutFinish =?= UNDEFINED) ||\
                 (StageOutFinish == 0))
```

on_exit_hold = <**ClassAd Boolean Expression**> The ClassAd expression is checked when the job exits, and if `True`, places the job into the Hold state. If `False` (the default value when not defined), then nothing happens and the `on_exit_remove` expression is checked to determine if that needs to be applied.

For example: Suppose a job is known to run for a minimum of an hour. If the job exits after less than an hour, the job should be placed on hold and an e-mail notification sent, instead of being allowed to leave the queue.

```
on_exit_hold = (CurrentTime - JobStartDate) < (60 * $(MINUTE))
```

This expression places the job on hold if it exits for any reason before running for an hour. An e-mail will be sent to the user explaining that the job was placed on hold because this expression became `True`.

`periodic_*` expressions take precedence over `on_exit_*` expressions, and `*_hold` expressions take precedence over a `*_remove` expressions.

Only job ClassAd attributes will be defined for use by this ClassAd expression. This expression is available for the vanilla, java, parallel, grid, local and scheduler universes. It is additionally available, when submitted from a Unix machine, for the standard universe.

on_exit_remove = <**ClassAd Boolean Expression**> The ClassAd expression is checked when the job exits, and if `True` (the default value when undefined), then it allows the job to leave the queue normally. If `False`, then the job is placed back into the Idle state. If the user job runs under the vanilla universe, then the job restarts from the beginning. If the user job runs under the standard universe, then it continues from where it left off, using the last checkpoint.

For example, suppose a job occasionally segfaults, but chances are that the job will finish successfully if the job is run again with the same data. The **on_exit_remove** expression can cause the job to run again with the following command. Assume that the signal identifier for the segmentation fault is 11 on the platform where the job will be running.

```
on_exit_remove = (ExitBySignal == False) || (ExitSignal != 11)
```

This expression lets the job leave the queue if the job was not killed by a signal or if it was killed by a signal other than 11, representing segmentation fault in this example. So, if the exited due to signal 11, it will stay in the job queue. In any other case of the job exiting, the job will leave the queue as it normally would have done.

As another example, if the job should only leave the queue if it exited on its own with status 0, this **on_exit_remove** expression works well:

```
on_exit_remove = (ExitBySignal == False) && (ExitCode == 0)
```

If the job was killed by a signal or exited with a non-zero exit status, Condor would leave the job in the queue to run again.

`periodic_*` expressions take precedence over `on_exit_*` expressions, and `*_hold` expressions take precedence over a `*_remove` expressions.

Only job ClassAd attributes will be defined for use by this ClassAd expression. This expression is available for the vanilla, java, parallel, grid, local and scheduler universes. It is additionally available, when submitted from a Unix machine, for the standard universe. Note that the *condor_schedd* daemon, by default, only checks these periodic expressions once every 300 seconds. The period of these evaluations can be adjusted by setting the `PERIODIC_EXPR_INTERVAL` configuration macro.

periodic_hold = <**ClassAd Boolean Expression**> This expression is checked periodically at an interval of the number of seconds set by the configuration variable `PERIODIC_EXPR_INTERVAL`. If it becomes `True`, the job will be placed on hold. If unspecified, the default value is `False`.

`periodic_*` expressions take precedence over `on_exit_*` expressions, and `*_hold` expressions take precedence over a `*_remove` expressions.

Only job ClassAd attributes will be defined for use by this ClassAd expression. This expression is available for the vanilla, java, parallel, grid, local and scheduler universes. It is additionally available, when submitted from a Unix machine, for the standard universe. Note that the *condor_schedd* daemon, by default, only checks these periodic expressions once every 300 seconds. The period of these evaluations can be adjusted by setting the `PERIODIC_EXPR_INTERVAL` configuration macro.

periodic_release = <**ClassAd Boolean Expression**> This expression is checked periodically at an interval of the number of seconds set by the configuration variable `PERIODIC_EXPR_INTERVAL` while the job is in the Hold state. If the expression becomes `True`, the job will be released.

Only job ClassAd attributes will be defined for use by this ClassAd expression. This expression is available for the vanilla, java, parallel, grid, local and scheduler universes. It is additionally available, when submitted from a Unix machine, for the standard universe. Note that the *condor_schedd* daemon, by default, only checks periodic expressions once every 300 seconds. The period of these evaluations can be adjusted by setting the `PERIODIC_EXPR_INTERVAL` configuration macro.

periodic_remove = <**ClassAd Boolean Expression**> This expression is checked periodically at an interval of the number of seconds set by the configuration variable `PERIODIC_EXPR_INTERVAL`. If it becomes `True`, the job is removed from the queue. If unspecified, the default value is `False`.

See section 9, the Examples section of the *condor_submit* manual page, for an example of a **periodic_remove** expression.

`periodic_*` expressions take precedence over `on_exit_*` expressions, and `*_hold` expressions take precedence over a `*_remove` expressions. So, the `periodic_remove` expression takes precedent over the `on_exit_remove` expression, if the two describe conflicting actions.

Only job ClassAd attributes will be defined for use by this ClassAd expression. This expression is available for the vanilla, java, parallel, grid, local and scheduler universes. It is additionally available, when submitted from a Unix machine, for the standard universe. Note that the *condor_schedd* daemon, by default, only checks periodic expressions once every 300 seconds. The period of these evaluations can be adjusted by setting the `PERIODIC_EXPR_INTERVAL` configuration macro.

next_job_start_delay = <**ClassAd Boolean Expression**> This expression specifies the number of seconds to delay after starting up this job before the next job is started. The maximum allowed delay is specified by the Condor configuration variable

MAX_NEXT_JOB_START_DELAY , which defaults to 10 minutes. This command does not apply to **scheduler** or **local** universe jobs.

This command has been historically used to implement a form of job start throttling from the job submitter's perspective. It was effective for the case of multiple job submission where the transfer of extremely large input data sets to the execute machine caused machine performance to suffer. This command is no longer useful, as throttling should be accomplished through configuration of the *condor_schedd* daemon.

COMMANDS SPECIFIC TO THE STANDARD UNIVERSE

allow_startup_script = <**True** | **False**> If True, a standard universe job will execute a script instead of submitting the job, and the consistency check to see if the executable has been linked using *condor_compile* is omitted. The **executable** command within the submit description file specifies the name of the script. The script is used to do preprocessing before the job is submitted. The shell script ends with an *exec* of the job executable, such that the process id of the executable is the same as that of the shell script. Here is an example script that gets a copy of a machine-specific executable before the *exec*.

```
#! /bin/sh

# get the host name of the machine
$host=`uname -n`

# grab a standard universe executable designed specifically
# for this host
scp elsewhere@cs.wisc.edu:${host} executable

# The PID MUST stay the same, so exec the new standard universe process.
exec executable ${1+"$@"}
```

If this command is not present (defined), then the value defaults to false.

append_files = file1, file2, ... If your job attempts to access a file mentioned in this list, Condor will force all writes to that file to be appended to the end. Furthermore, condor_submit will not truncate it. This list uses the same syntax as compress_files, shown above.

This option may yield some surprising results. If several jobs attempt to write to the same file, their output may be intermixed. If a job is evicted from one or more machines during the course of its lifetime, such an output file might contain several copies of the results. This option should be only be used when you wish a certain file to be treated as a running log instead of a precise result.

This option only applies to standard-universe jobs.

buffer_files = < **" name** = **(size,block-size) ; name2** = **(size,block-size) ... "** >

buffer_size = <**bytes-in-buffer**>

buffer_block_size = <**bytes-in-block**> Condor keeps a buffer of recently-used data for each file a job accesses. This buffer is used both to cache commonly-used data and to consolidate small reads and writes into larger operations that get better throughput. The default settings should produce reasonable results for most programs.

These options only apply to standard-universe jobs.

If needed, you may set the buffer controls individually for each file using the buffer_files option. For example, to set the buffer size to 1 Mbyte and the block size to 256 Kbytes for the file `input.data`, use this command:

```
buffer_files = "input.data=(1000000,256000)"
```

Alternatively, you may use these two options to set the default sizes for all files used by your job:

```
buffer_size = 1000000
buffer_block_size = 256000
```

If you do not set these, Condor will use the values given by these two configuration file macros:

```
DEFAULT_IO_BUFFER_SIZE = 1000000
DEFAULT_IO_BUFFER_BLOCK_SIZE = 256000
```

Finally, if no other settings are present, Condor will use a buffer of 512 Kbytes and a block size of 32 Kbytes.

compress_files = file1, file2, ... If your job attempts to access any of the files mentioned in this list, Condor will automatically compress them (if writing) or decompress them (if reading). The compress format is the same as used by GNU gzip.

The files given in this list may be simple file names or complete paths and may include $*$ as a wild card. For example, this list causes the file /tmp/data.gz, any file named event.gz, and any file ending in .gzip to be automatically compressed or decompressed as needed:

```
compress_files = /tmp/data.gz, event.gz, *.gzip
```

Due to the nature of the compression format, compressed files must only be accessed sequentially. Random access reading is allowed but is very slow, while random access writing is simply not possible. This restriction may be avoided by using both compress_files and fetch_files at the same time. When this is done, a file is kept in the decompressed state at the execution machine, but is compressed for transfer to its original location.

This option only applies to standard universe jobs.

fetch_files = file1, file2, ... If your job attempts to access a file mentioned in this list, Condor will automatically copy the whole file to the executing machine, where it can be accessed quickly. When your job closes the file, it will be copied back to its original location. This list uses the same syntax as compress_files, shown above.

This option only applies to standard universe jobs.

file_remaps = < **“ name** = **newname ; name2** = **newname2 ... ”**> Directs Condor to use a new file name in place of an old one. *name* describes a file name that your job may attempt to open, and *newname* describes the file name it should be replaced with. *newname* may include an optional leading access specifier, `local:` or `remote:`. If left unspecified, the default access specifier is `remote:`. Multiple remaps can be specified by separating each with a semicolon.

This option only applies to standard universe jobs.

If you wish to remap file names that contain equals signs or semicolons, these special characters may be escaped with a backslash.

Example One: Suppose that your job reads a file named `dataset.1`. To instruct Condor to force your job to read `other.dataset` instead, add this to the submit file:

```
file_remaps = "dataset.1=other.dataset"
```

Example Two: Suppose that your run many jobs which all read in the same large file, called `very.big`. If this file can be found in the same place on a local disk in every machine in the pool, (say `/bigdisk/bigfile`,) you can instruct Condor of this fact by remapping `very.big` to `/bigdisk/bigfile` and specifying that the file is to be read locally, which will be much faster than reading over the network.

```
file_remaps = "very.big = local:/bigdisk/bigfile"
```

Example Three: Several remaps can be applied at once by separating each with a semicolon.

```
file_remaps = "very.big = local:/bigdisk/bigfile ; dataset.1 = other.dataset"
```

local_files = file1, file2, ... If your job attempts to access a file mentioned in this list, Condor will cause it to be read or written at the execution machine. This is most useful for temporary files not used for input or output. This list uses the same syntax as compress_files, shown above.

```
local_files = /tmp/*
```

This option only applies to standard universe jobs.

want_remote_io = <**True** | **False**> This option controls how a file is opened and manipulated in a standard universe job. If this option is true, which is the default, then the *condor_shadow* makes all decisions about how each and every file should be opened by the executing job. This entails a network round trip (or more) from the job to the *condor_shadow* and back again for every single `open()` in addition to other needed information about the file. If set to false, then when the job queries the *condor_shadow* for the first time about how to open a file, the *condor_shadow* will inform the job to automatically perform all of its file manipulation on the local file system on the execute machine and any file remapping will be ignored. This means that there **must** be a shared file system (such as NFS or AFS) between the execute machine and the submit machine and that **ALL** paths that the job could open on the execute machine must be valid. The ability of the standard universe job to checkpoint, possibly to a checkpoint server, is not affected by this attribute. However, when the job resumes it will be expecting the same file system conditions that were present when the job checkpointed.

COMMANDS FOR THE GRID

amazon_ami_id = <**Amazon EC2 AMI ID**> AMI identifier of the VM image to run for **amazon** jobs.

amazon_instance_type = <**VM Type**> An identifier for the type of VM desired for an **amazon** job. The default value is `m1.small`. Other values are `m1.large`, `m1.xlarge`, `c1.medium`, and `c1.xlarge`.

amazon_keypair_file = <**pathname**> The complete path and filename of a file into which Condor will write an ssh key for use with **amazon** jobs. The key can be used to ssh into the virtual machine once it is running.

amazon_private_key = <**pathname**> Used for **amazon** jobs. Path and filename of a file containing the private key to be used to authenticate with Amazon's EC2 service via SOAP.

amazon_public_key = <**pathname**> Used for **amazon** jobs. Path and filename of a file containing the public X509 certificate to be used to authenticate with Amazon's EC2 service via SOAP.

amazon_security_groups = group1, group2, ... Used for **amazon** jobs. A list of Amazon EC2 security group names, which should be associated with the job.

amazon_user_data = <**data**> Used for **amazon** jobs. A block of data that can be accessed by the virtual machine job inside Amazon EC2. If both **amazon_user_data** and **amazon_user_data_file** are provided, the two blocks of data are concatenated, with the data from **amazon_user_data** occurring first.

amazon_user_data_file = <**pathname**> Used for **amazon** jobs. A file containing data that can be accessed by the virtual machine job inside Amazon EC2. If both **amazon_user_data** and **amazon_user_data_file** are provided, the two blocks of data are concatenated, with the data from **amazon_user_data** occurring first.

cream_attributes = <**name=value;**. . .**;name=value**> Provides a list of attribute/value pairs to be set in a CREAM job description of a grid universe job destined for the CREAM grid system. The pairs are separated by semicolons, and written in New ClassAd syntax.

deltacloud_hardware_profile = <**Deltacloud profile name**> Used for **deltacloud** jobs. An optional identifier for the type of VM desired. If not provided, a service-defined default is used.

deltacloud_hardware_profile_cpu = <**cpu details**> Used for **deltacloud** jobs. An optional description of the CPUs desired for the VM, overriding the selected hardware profile.

deltacloud_hardware_profile_memory = <**memory details**> Used for **deltacloud** jobs. An optional description of the memory (RAM) desired for the VM, overriding the selected hardware profile.

deltacloud_hardware_profile_storage = <**storage details**> Used for **deltacloud** jobs. An optional description of the storage (disk) desired for the VM, overriding the selected hardware profile.

deltacloud_image_id = <**Deltacloud image ID**> Used for **deltacloud** jobs. Identifier of the VM image to run.

deltacloud_keyname = <**Deltacloud key name**> Used for **deltacloud** jobs. Identifier of the SSH key pair that should be used to allow remote login to the running instance. The key pair needs to be created before submission.

deltacloud_password_file = <**pathname**> Used for **deltacloud** jobs. Path and file name of a file containing the secret key to be used to authenticate with a Deltacloud service.

deltacloud_realm_id = <**Deltacloud realm ID**> Used for **deltacloud** jobs. An optional identifier specifying which of multiple locations within a cloud service should be used to run the VM. If not provided, a service-selected default is used.

deltacloud_user_data = <**data**> Used for **deltacloud** jobs. A string, representing a block of data that can be accessed by the virtual machine job inside the cloud service.

deltacloud_username = <**Deltacloud username**> Used for **deltacloud** jobs. The user name to be used to authenticate with a Deltacloud service.

globus_rematch = <**ClassAd Boolean Expression**> This expression is evaluated by the *condor_gridmanager* whenever:

1. the **globus_resubmit** expression evaluates to `True`
2. the *condor_gridmanager* decides it needs to retry a submission (as when a previous submission failed to commit)

If **globus_rematch** evaluates to `True`, then *before* the job is submitted again to globus, the *condor_gridmanager* will request that the *condor_schedd* daemon renegotiate with the matchmaker (the *condor_negotiator*). The result is this job will be matched again.

globus_resubmit = <**ClassAd Boolean Expression**> The expression is evaluated by the *condor_gridmanager* each time the *condor_gridmanager* gets a job ad to manage. Therefore, the expression is evaluated:

1. when a grid universe job is first submitted to Condor-G
2. when a grid universe job is released from the hold state
3. when Condor-G is restarted (specifically, whenever the *condor_gridmanager* is restarted)

If the expression evaluates to `True`, then any previous submission to the grid universe will be forgotten and this job will be submitted again as a fresh submission to the grid universe. This may be useful if there is a desire to give up on a previous submission and try again. Note that this may result in the same job running more than once. Do not treat this operation lightly.

globus_rsl = <**RSL-string**> Used to provide any additional Globus RSL string attributes which are not covered by other submit description file commands or job attributes. Used for **grid universe** jobs, where the grid resource has a **grid-type-string** of **gt2**.

globus_xml = <**XML-string**> Used to provide any additional attributes in the GRAM XML job description that Condor writes which are not covered by regular submit description file parameters. Used for grid type **gt4** jobs.

grid_resource = <**grid-type-string**> <**grid-specific-parameter-list**> For each **grid-type-string** value, there are further type-specific values that must specified. This submit description file command allows each to be given in a space-separated list. Allowable **grid-type-string** values are **amazon**, **condor**, **cream**, **deltacloud**, **gt2**, **gt4**, **gt5**, **lsf**, **nordugrid**, **pbs**, and **unicore**. See section 5.3 for details on the variety of grid types.

For a **grid-type-string** of **amazon**, one additional parameter specifies the EC2 URL. See section 5.3.7 for details.

For a **grid-type-string** of **condor**, the first parameter is the name of the remote *condor_schedd* daemon. The second parameter is the name of the pool to which the remote *condor_schedd* daemon belongs. See section 5.3.1 for details.

For a **grid-type-string** of **cream**, there are three parameters. The first parameter is the web services address of the CREAM server. The second parameter is the name of the batch system that sits behind the CREAM server. The third parameter identifies a site-specific queue within the batch system. See section 5.3.8 for details.

For a **grid-type-string** of **deltacloud**, the single parameter is the URL of the deltacloud service requested. See section 5.3.9 for details.

For a **grid-type-string** of **gt2**, the single parameter is the name of the pre-WS GRAM resource to be used. See section 5.3.2 for details.

For a **grid-type-string** of **gt4**, the first parameter is the name of the WS GRAM service to be used. The second parameter is the name of WS resource to be used (usually the name of the back-end scheduler). See section 5.3.2 for details.

For a **grid-type-string** of **gt5**, the single parameter is the name of the pre-WS GRAM resource to be used, which is the same as for the **grid-type-string** of **gt2**. See section 5.3.2 for details.

For a **grid-type-string** of **lsf**, no additional parameters are used. See section 5.3.6 for details.

For a **grid-type-string** of **nordugrid**, the single parameter is the name of the NorduGrid resource to be used. See section 5.3.3 for details.

For a **grid-type-string** of **pbs**, no additional parameters are used. See section 5.3.5 for details.

For a **grid-type-string** of **unicore**, the first parameter is the name of the Unicore Usite to be used. The second parameter is the name of the Unicore Vsite to be used. See section 5.3.4 for details.

keystore_alias = <**name**> A string to locate the certificate in a Java keystore file, as used for a **unicore** job.

keystore_file = <**pathname**> The complete path and file name of the Java keystore file containing the certificate to be used for a **unicore** job.

keystore_passphrase_file = <**pathname**> The complete path and file name to the file containing the passphrase protecting a Java keystore file containing the certificate. Relevant for a **unicore** job.

MyProxyCredentialName = <**symbolic name**> The symbolic name that identifies a credential to the *MyProxy* server. This symbolic name is set as the credential is initially stored on the server (using *myproxy-init*).

MyProxyHost = <**host**>**:**<**port**> The Internet address of the host that is the *MyProxy* server. The **host** may be specified by either a host name (as in `head.example.com`) or an IP address (of the form 123.456.7.8). The **port** number is an integer.

MyProxyNewProxyLifetime = <**number-of-minutes**> The new lifetime (in minutes) of the proxy after it is refreshed.

MyProxyPassword = <**password**> The password needed to refresh a credential on the *MyProxy* server. This password is set when the user initially stores credentials on the server (using *myproxy-init*). As an alternative to using **MyProxyPassword** in the submit description file, the password may be specified as a command line argument to *condor_submit* with the *-password* argument.

MyProxyRefreshThreshold = <**number-of-seconds**> The time (in seconds) before the expiration of a proxy that the proxy should be refreshed. For example, if **MyProxyRefreshThreshold** is set to the value 600, the proxy will be refreshed 10 minutes before it expires.

MyProxyServerDN = <**credential subject**> A string that specifies the expected Distinguished Name (credential subject, abbreviated DN) of the *MyProxy* server. It must be specified when the *MyProxy* server DN does not follow the conventional naming scheme of a host credential. This occurs, for example, when the *MyProxy* server DN begins with a user credential.

nordugrid_rsl = <**RSL-string**> Used to provide any additional RSL string attributes which are not covered by regular submit description file parameters. Used when the **universe** is **grid**, and the type of grid system is **nordugrid**.

transfer_error = <**True** | **False**> For jobs submitted to the grid universe only. If `True`, then the error output (from `stderr`) from the job is transferred from the remote machine back to the submit machine. The name of the file after transfer is given by the **error** command. If `False`, no transfer takes place (from the remote machine to submit machine), and the name of the file is given by the **error** command. The default value is `True`.

transfer_input = <**True** | **False**> For jobs submitted to the grid universe only. If `True`, then the job input (`stdin`) is transferred from the machine where the job was submitted to the remote machine. The name of the file that is transferred is given by the **input** command. If `False`, then the job's input is taken from a pre-staged file on the remote machine, and the name of the file is given by the **input** command. The default value is `True`.

For transferring files other than `stdin`, see **transfer_input_files**.

transfer_output = <**True** | **False**> For jobs submitted to the grid universe only. If `True`, then the output (from `stdout`) from the job is transferred from the remote machine back to the submit machine. The name of the file after transfer is given by the **output** command. If `False`, no transfer takes place (from the remote machine to submit machine), and the name of the file is given by the **output** command. The default value is `True`.

For transferring files other than `stdout`, see **transfer_output_files**.

x509userproxy = <**full-pathname**> Used to override the default path name for X.509 user certificates. The default location for X.509 proxies is the `/tmp` directory, which is generally a local file system. Setting this value would allow Condor to access the proxy in a shared file system (for example, AFS). Condor will use the proxy specified in the submit description file first. If nothing is specified in the submit description file, it will use the environment variable X509_USER_CERT. If that variable is not present, it will search in the default location.

x509userproxy is relevant when the **universe** is **vanilla**, or when the **universe** is **grid** and the type of grid system is one of **gt2**, **gt4**, or **nordugrid**. Defining a value causes the proxy to be delegated to the execute machine. Further, VOMS attributes defined in the proxy will appear in the job ClassAd. See the unnumbered subsection labeled Job ClassAd Attributes on page 902 for all job attribute descriptions.

delegate_job_GSI_credentials_lifetime = <**seconds**> This command specifies the maximum number of seconds for which delegated proxies should be valid. The default behavior is determined by the configuration setting `DELEGATE_JOB_GSI_CREDENTIALS_LIFETIME`, which defaults to one day. A value of 0 indicates that the delegated proxy should be valid for as long as allowed by the credential used to create the proxy. This setting currently only applies to proxies delegated for non-grid jobs and Condor-C jobs. It does not currently apply to globus grid jobs, which always behave as though this setting were 0. This setting has no effect if the configuration setting `DELEGATE_JOB_GSI_CREDENTIALS` is false, because in that case the job proxy is copied rather than delegated.

COMMANDS FOR PARALLEL, JAVA, and SCHEDULER UNIVERSES

hold_kill_sig = <**signal-number**> For the scheduler universe only, **signal-number** is the signal delivered to the job when the job is put on hold with *condor_hold*. **signal-number** may be either the platform-specific name or value of the signal. If this command is not present, the value of **kill_sig** is used.

jar_files = <**file_list**> Specifies a list of additional JAR files to include when using the Java universe. JAR files will be transferred along with the executable and automatically added to the classpath.

java_vm_args = <**argument_list**> Specifies a list of additional arguments to the Java VM itself, When Condor runs the Java program, these are the arguments that go before the class name. This can be used to set VM-specific arguments like stack size, garbage-collector arguments and initial property values.

machine_count = <**max**> For the parallel universe, a single value (*max*) is required. It is neither a maximum or minimum, but the number of machines to be dedicated toward running the job.

remove_kill_sig = <**signal-number**> For the scheduler universe only, **signal-number** is the signal delivered to the job when the job is removed with *condor_rm*. **signal-number** may be either the platform-specific name or value of the signal. This example shows it both ways for a Linux signal:

```
remove_kill_sig = SIGUSR1
remove_kill_sig = 10
```

If this command is not present, the value of **kill_sig** is used.

COMMANDS FOR THE VM UNIVERSE

kvm_transfer_files = <**list-of-files**> A comma separated list of all files that Condor is to transfer to the execute machine.

vm_disk = file1:device1:permission1, file2:device2:permission2:format2, ... A list of comma separated disk files. Each disk file is specified by 4 colon separated fields. The first field is the path and file name of the disk file. The second field specifies the device. The third field specifies permissions, and the optional fourth field specifies the image format.

An example that specifies two disk files:

```
vm_disk = /myxen/diskfile.img:sda1:w,/myxen/swap.img:sda2:w
```

vm_checkpoint = <**True** | **False**> A boolean value specifying whether or not to take checkpoints. If not specified, the default value is `False`. In the current implementation, setting both **vm_checkpoint** and **vm_networking** to `True` does not yet work in all cases. Networking cannot be used if a vm universe job uses a checkpoint in order to continue execution after migration to another machine.

vm_macaddr = <**MACAddr**> Defines that MAC address that the virtual machine's network interface should have, in the standard format of six groups of two hexadecimal digits separated by colons.

vm_memory = <**MBytes-of-memory**> The amount of memory in MBytes that a vm universe job requires.

vm_networking = <**True** | **False**> Specifies whether to use networking or not. In the current implementation, setting both **vm_checkpoint** and **vm_networking** to `True` does not yet work in all cases. Networking cannot be used if a vm universe job uses a checkpoint in order to continue execution after migration to another machine.

vm_networking_type = <**nat** | **bridge** > When **vm_networking** is `True`, this definition augments the job's requirements to match only machines with the specified networking. If not specified, then either networking type matches.

vm_no_output_vm = <**True** | **False**> When `True`, prevents Condor from transferring output files back to the machine from which the vm universe job was submitted. If not specified, the default value is `False`.

vm_type = <**vmware** | **xen** | **kvm**> Specifies the underlying virtual machine software that this job expects.

vmware_dir = <**pathname**> The complete path and name of the directory where VMware-specific files and applications such as the VMDK (Virtual Machine Disk Format) and VMX (Virtual Machine Configuration) reside. This command is optional; when not specified, all relevant VMware image files are to be listed using **transfer_input_files**.

vmware_should_transfer_files = <**True** | **False**> Specifies whether Condor will transfer VMware-specific files located as specified by **vmware_dir** to the execute machine (`True`) or rely on access through a shared file system (`False`). Omission of this required command (for VMware vm universe jobs) results in an error message from *condor_submit*, and the job will not be submitted.

vmware_snapshot_disk = <**True** | **False**> When `True`, causes Condor to utilize a VMware snapshot disk for new or modified files. If not specified, the default value is `True`.

xen_initrd = <**image-file**> When **xen_kernel** gives a path and file name for the kernel image to use, this optional command may specify a path to and ramdisk (`initrd`) image file.

xen_kernel = <**included** | **path-to-kernel**> A value of **included** specifies that the kernel is included in the disk file. If not one of these values, then the value is a path and file name of the kernel to be used.

xen_kernel_params = <**string**> A string that is appended to the Xen kernel command line.

xen_root = <**string**> A string that is appended to the Xen kernel command line to specify the root device. This string is required when **xen_kernel** gives a path to a kernel. Omission for this required case results in an error message during submission.

xen_transfer_files = <**list-of-files**> A comma separated list of all files that Condor is to transfer to the execute machine.

ADVANCED COMMANDS

concurrency_limits = <**string-list**> A list of resources that this job needs. The resources are presumed to have concurrency limits placed upon them, thereby limiting the number of concurrent jobs in execution which need the named resource. Commas and space characters delimit the items in the list. Each item in the list may specify a numerical value identifying the integer number of resources required for the job. The syntax follows the resource name by a colon character (`:`) and the numerical value. See section 3.13.14 for details on concurrency limits.

copy_to_spool = <**True** | **False**> If **copy_to_spool** is `True`, then *condor_submit* copies the executable to the local spool directory before running it on a remote host. As copying can be quite time consuming and unnecessary, the default value is `False` for all job universes other than the standard universe. When `False`, *condor_submit* does not copy the executable to a local spool directory. The default is `True` in standard universe, because resuming execution from a checkpoint can only be guaranteed to work using precisely the same executable that created the checkpoint.

coresize = <**size**> Should the user's program abort and produce a core file, **coresize** specifies the maximum size in bytes of the core file which the user wishes to keep. If **coresize** is not specified in the command file, the system's user resource limit `coredumpsize` is used. A value of -1 results in no limits being applied to the core file size.

cron_day_of_month = <**Cron-evaluated Day**> The set of days of the month for which a deferral time applies. See section 2.12.2 for further details and examples.

cron_day_of_week = <**Cron-evaluated Day**> The set of days of the week for which a deferral time applies. See section 2.12.2 for details, semantics, and examples.

cron_hour = <**Cron-evaluated Hour**> The set of hours of the day for which a deferral time applies. See section 2.12.2 for details, semantics, and examples.

cron_minute = <**Cron-evaluated Minute**> The set of minutes within an hour for which a deferral time applies. See section 2.12.2 for details, semantics, and examples.

cron_month = <**Cron-evaluated Month**> The set of months within a year for which a deferral time applies. See section 2.12.2 for details, semantics, and examples.

cron_prep_time = <**ClassAd Integer Expression**> Analogous to **deferral_prep_time**. The number of seconds prior to a job's deferral time that the job may be matched and sent to an execution machine.

cron_window = <**ClassAd Integer Expression**> Analogous to the submit command **deferral_window**. It allows cron jobs that miss their deferral time to begin execution.

See section 2.12.1 for further details and examples.

deferral_prep_time = <**ClassAd Integer Expression**> The number of seconds prior to a job's deferral time that the job may be matched and sent to an execution machine.

See section 2.12.1 for further details.

deferral_time = <**ClassAd Integer Expression**> Allows a job to specify the time at which its execution is to begin, instead of beginning execution as soon as it arrives at the execution machine. The deferral time is an expression that evaluates to a Unix Epoch timestamp (the number of seconds elapsed since 00:00:00 on January 1, 1970, Coordinated Universal Time). Deferral time is evaluated with respect to the execution machine. This option delays the start of execution, but not the matching and claiming of a machine for the job. If the job is not available and ready to begin execution at the deferral time, it has missed its deferral time. A job that misses its deferral time will be put on hold in the queue.

See section 2.12.1 for further details and examples.

Due to implementation details, a deferral time may not be used for scheduler universe jobs.

deferral_window = <**ClassAd Integer Expression**> The deferral window is used in conjunction with the **deferral_time** command to allow jobs that miss their deferral time to begin execution.

See section 2.12.1 for further details and examples.

email_attributes = <**list-of-job-ad-attributes**> A comma-separated list of attributes from the job ClassAd. These attributes and their values will be included in the e-mail notification of job completion.

image_size = <**size**> Advice to Condor specifying the maximum virtual image size to which the job will grow during its execution. Condor will then execute the job only on machines which have enough resources, (such as virtual memory), to support executing the job. If not specified, Condor will automatically make a (reasonably accurate) estimate about the job's size and adjust this estimate as the program runs. If specified and underestimated, the job may crash

due to the inability to acquire more address space; for example, if `malloc()` fails. If the image size is overestimated, Condor may have difficulty finding machines which have the required resources. *size* is specified in Kbytes. For example, for an image size of 8 Megabytes, *size* should be 8000.

initialdir = <**directory-path**> Used to give jobs a directory with respect to file input and output. Also provides a directory (on the machine from which the job is submitted) for the user log, when a full path is not specified.

For vanilla universe jobs where there is a shared file system, it is the current working directory on the machine where the job is executed.

For vanilla or grid universe jobs where file transfer mechanisms are utilized (there is *not* a shared file system), it is the directory on the machine from which the job is submitted where the input files come from, and where the job's output files go to.

For standard universe jobs, it is the directory on the machine from which the job is submitted where the *condor_shadow* daemon runs; the current working directory for file input and output accomplished through remote system calls.

For scheduler universe jobs, it is the directory on the machine from which the job is submitted where the job runs; the current working directory for file input and output with respect to relative path names.

Note that the path to the executable is *not* relative to **initialdir**; if it is a relative path, it is relative to the directory in which the *condor_submit* command is run.

job_ad_information_attrs = <**attribute-list**> A comma-separated list of job ClassAd attribute names. The named attributes and their values are written to the user log whenever any event is being written to the log. This implements the same thing as the configuration variable `EVENT_LOG_INFORMATION_ATTRS` (see page 175), but it applies to the user log, instead of the system event log.

job_lease_duration = <**number-of-seconds**> For vanilla and java universe jobs only, the duration (in seconds) of a job lease. The default value is twenty minutes for universes that support it. If a job lease is not desired, the value can be explicitly set to 0 to disable the job lease semantics. See section 2.14.4 for details of job leases.

job_machine_attrs = <**attr1, attr2,** . . .> A comma and/or space separated list of machine attribute names that should be recorded in the job ClassAd in addition to the ones specified by the *condor_schedd* daemon's system configuration variable `SYSTEM_JOB_MACHINE_ATTRS` . When there are multiple run attempts, history of machine attributes from previous run attempts may be kept. The number of run attempts to store may be extended beyond the system-specified history length by using the submit file command **job_machine_attrs_history_length**. A machine attribute named `X` will be inserted into the job ClassAd as an attribute named `MachineAttrX0`. The previous value of this attribute will be named `MachineAttrX1`, the previous to that will be named `MachineAttrX2`, and so on, up to the specified history length. A history of length 1 means that only `MachineAttrX0` will be recorded. The value recorded in the job ClassAd is the evaluation of the machine attribute in the context of the job ClassAd when the *condor_schedd*

daemon initiates the start up of the job. If the evaluation results in an `Undefined` or `Error` result, the value recorded in the job ad will be `Undefined` or `Error`, respectively.

kill_sig = <**signal-number**> When Condor needs to kick a job off of a machine, it will send the job the signal specified by **signal-number**. **signal-number** needs to be an integer which represents a valid signal on the execution machine. For jobs submitted to the standard universe, the default value is the number for `SIGTSTP` which tells the Condor libraries to initiate a checkpoint of the process. For jobs submitted to other universes, the default value, when not defined, is `SIGTERM`, which is the standard way to terminate a program in Unix.

kill_sig_timeout = <**seconds**> The number of seconds that Condor should wait following the sending of the kill signal defined by **kill_sig** and forcibly killing the job. The actual amount of time between sending the signal and forcibly killing the job is the smallest of this value and the configuration variable `KILLING_TIMEOUT` , as defined on the execute machine.

load_profile = <**True** | **False**> When `True`, loads the account profile of the dedicated run account for Windows jobs. May not be used with **run_as_owner**.

match_list_length = <**integer value**> Defaults to the value zero (0). When **match_list_length** is defined with an integer value greater than zero (0), attributes are inserted into the job ClassAd. The maximum number of attributes defined is given by the integer value. The job ClassAds introduced are given as

```
LastMatchName0 = "most-recent-Name"
LastMatchName1 = "next-most-recent-Name"
```

The value for each introduced ClassAd is given by the value of the `Name` attribute from the machine ClassAd of a previous execution (match). As a job is matched, the definitions for these attributes will roll, with `LastMatchName1` becoming `LastMatchName2`, `LastMatchName0` becoming `LastMatchName1`, and `LastMatchName0` being set by the most recent value of the `Name` attribute.

An intended use of these job attributes is in the requirements expression. The requirements can allow a job to prefer a match with either the same or a different resource than a previous match.

max_job_retirement_time = <**integer expression**> An integer-valued expression (in seconds) that does nothing unless the machine that runs the job has been configured to provide retirement time (see section 3.5.8). Retirement time is a grace period given to a job to finish naturally when a resource claim is about to be preempted. No kill signals are sent during a retirement time. The default behavior in many cases is to take as much retirement time as the machine offers, so this command will rarely appear in a submit description file.

When a resource claim is to be preempted, this expression in the submit file specifies the maximum run time of the job (in seconds, since the job started). This expression has no effect, if it is greater than the maximum retirement time provided by the machine policy. If the resource claim is *not* preempted, this expression and the machine retirement policy are irrelevant. If the resource claim *is* preempted and the job finishes sooner than the maximum time, the claim closes gracefully and all is well. If the resource claim is preempted and the

job does *not* finish in time, the usual preemption procedure is followed (typically a soft kill signal, followed by some time to gracefully shut down, followed by a hard kill signal).

Standard universe jobs and any jobs running with **nice_user** priority have a default **max_job_retirement_time** of 0, so no retirement time is utilized by default. In all other cases, no default value is provided, so the maximum amount of retirement time is utilized by default.

Setting this expression does not affect the job's resource requirements or preferences. For a job to only run on a machine with a minimum , or to preferentially run on such machines, explicitly specify this in the requirements and/or rank expressions.

nice_user = <**True** | **False**> Normally, when a machine becomes available to Condor, Condor decides which job to run based upon user and job priorities. Setting **nice_user** equal to `True` tells Condor not to use your regular user priority, but that this job should have last priority among all users and all jobs. So jobs submitted in this fashion run only on machines which no other non-nice_user job wants — a true "bottom-feeder" job! This is very handy if a user has some jobs they wish to run, but do not wish to use resources that could instead be used to run other people's Condor jobs. Jobs submitted in this fashion have "nice-user." pre-appended in front of the owner name when viewed from *condor_q* or *condor_userprio*. The default value is **False**.

noop_job = <**ClassAd Boolean Expression**> When this boolean expression is `True`, the job is immediately removed from the queue, and Condor makes no attempt at running the job. The log file for the job will show a job submitted event and a job terminated event, along with an exit code of 0, unless the user specifies a different signal or exit code.

noop_job_exit_code = <**return value**> When **noop_job** is in the submit description file and evaluates to `True`, this command allows the job to specify the return value as shown in the job's log file job terminated event. If not specified, the job will show as having terminated with status 0. This overrides any value specified with **noop_job_exit_signal**.

noop_job_exit_signal = <**signal number**> When **noop_job** is in the submit description file and evaluates to `True`, this command allows the job to specify the signal number that the job's log event will show the job having terminated with.

remote_initialdir = <**directory-path**> The path specifies the directory in which the job is to be executed on the remote machine. This is currently supported in all universes except for the standard universe.

rendezvousdir = <**directory-path**> Used to specify the shared file system directory to be used for file system authentication when submitting to a remote scheduler. Should be a path to a preexisting directory.

request_cpus = <**num-cpus**> For pools that enable dynamic *condor_startd* provisioning (see section 3.13.9), the number of CPUs requested for this job. If not specified, the number requested under dynamic *condor_startd* provisioning will be 1.

request_disk = <**quantity**> For pools that enable dynamic *condor_startd* provisioning (see section 3.13.9), the amount of disk space in Kbytes requested for this job, setting an initial value

for the job ClassAd attribute `DiskUsage`. If not specified, the initial amount requested under dynamic *condor_startd* provisioning will depend on the job universe. For **vm** universe jobs, it will be the size of the disk image. For other universes, it will be the sum of sizes of the job's executable and all input files.

request_memory = <**quantity**> For pools that enable dynamic *condor_startd* provisioning (see section 3.13.9), the amount of memory space in Mbytes requested for this job, setting an initial value for the job ClassAd attribute `ImageSize`. If not specified, the initial amount requested under dynamic *condor_startd* provisioning will depend on the job universe. For **vm** universe jobs that do not specify the request with **vm_memory**, it will be 0. For other universes, it will be the size of the job's executable.

run_as_owner = <**True** | **False**> A boolean value that causes the job to be run under the login of the submitter, if supported by the joint configuration of the submit and execute machines. On Unix platforms, this defaults to `True`, and on Windows platforms, it defaults to `False`. May not be used with **load_profile**. See section 6.2.4 for administrative details on configuring Windows to support this option, as well as section 3.3.7 on page 185 for a definition of `STARTER_ALLOW_RUNAS_OWNER`.

submit_event_notes = <**note**> A string that is appended to the submit event in the job's log file. For DAGMan jobs, the string `DAG Node:` and the node's name is automatically defined for **submit_event_notes**, causing the logged submit event to identify the DAG node job submitted.

+<**attribute**> = <**value**> A line which begins with a '+' (plus) character instructs *condor_submit* to insert the following *attribute* into the job ClassAd with the given *value*.

In addition to commands, the submit description file can contain macros and comments:

Macros Parameterless macros in the form of `$(macro_name)` may be inserted anywhere in Condor submit description files. Macros can be defined by lines in the form of

```
<macro_name> = <string>
```

Three pre-defined macros are supplied by the submit description file parser. The third of the pre-defined macros is only relevant to MPI applications under the parallel universe. The `$(Cluster)` macro supplies the value of the `ClusterId` job ClassAd attribute, and the `$(Process)` macro supplies the value of the `ProcId` job ClassAd attribute. These macros are intended to aid in the specification of input/output files, arguments, etc., for clusters with lots of jobs, and/or could be used to supply a Condor process with its own cluster and process numbers on the command line. The `$(Node)` macro is defined for MPI applications run as parallel universe jobs. It is a unique value assigned for the duration of the job that essentially identifies the machine on which a program is executing.

To use the dollar sign character (`$`) as a literal, without macro expansion, use

```
$(DOLLAR)
```

In addition to the normal macro, there is also a special kind of macro called a *substitution macro* that allows the substitution of a ClassAd attribute value defined on the resource machine itself (gotten after a match to the machine has been made) into specific commands within the submit description file. The substitution macro is of the form:

```
$$(attribute)
```

A common use of this macro is for the heterogeneous submission of an executable:

```
executable = povray.$$(opsys).$$(arch)
```

Values for the `opsys` and `arch` attributes are substituted at match time for any given resource. This allows Condor to automatically choose the correct executable for the matched machine.

An extension to the syntax of the substitution macro provides an alternative string to use if the machine attribute within the substitution macro is undefined. The syntax appears as:

```
$$(attribute:string_if_attribute_undefined)
```

An example using this extended syntax provides a path name to a required input file. Since the file can be placed in different locations on different machines, the file's path name is given as an argument to the program.

```
argument = $$(input_file_path:/usr/foo)
```

On the machine, if the attribute `input_file_path` is not defined, then the path `/usr/foo` is used instead.

A further extension to the syntax of the substitution macro allows the evaluation of a ClassAd expression to define the value. As all substitution macros, the expression is evaluated after a match has been made. Therefore, the expression may refer to machine attributes by prefacing them with the scope resolution prefix `TARGET.`, as specified in section 4.1.3. To place a ClassAd expression into the substitution macro, square brackets are added to delimit the expression. The syntax appears as:

```
$$([ClassAd expression])
```

An example of a job that uses this syntax may be one that wants to know how much memory it can use. The application cannot detect this itself, as it would potentially use all of the memory on a multi-slot machine. So the job determines the memory per slot, reducing it by 10% to account for miscellaneous overhead, and passes this as a command line argument to the application. In the submit description file will be

```
arguments=--memory $$([TARGET.Memory * 0.9])
```

To insert two dollar sign characters (`$$`) as literals into a ClassAd string, use

```
$$(DOLLARDOLLAR)
```

The environment macro, $ENV, allows the evaluation of an environment variable to be used in setting a submit description file command. The syntax used is

```
$ENV(variable)
```

An example submit description file command that uses this functionality evaluates the submittor's home directory in order to set the path and file name of a log file:

```
log = $ENV(HOME)/jobs/logfile
```

The environment variable is evaluated when the submit description file is processed.

The $RANDOM_CHOICE macro allows a random choice to be made from a given list of parameters at submission time. For an expression, if some randomness needs to be generated, the macro may appear as

```
$RANDOM_CHOICE(0,1,2,3,4,5,6)
```

When evaluated, one of the parameters values will be chosen.

Comments Blank lines and lines beginning with a pound sign ('#') character are ignored by the submit description file parser.

Exit Status

condor_submit will exit with a status value of 0 (zero) upon success, and a non-zero value upon failure.

Examples

- Submit Description File Example 1: This example queues three jobs for execution by Condor. The first will be given command line arguments of *15* and *2000*, and it will write its standard output to `foo.out1`. The second will be given command line arguments of *30* and *2000*, and it will write its standard output to `foo.out2`. Similarly the third will have arguments of *45* and *6000*, and it will use `foo.out3` for its standard output. Standard error output (if any) from all three programs will appear in `foo.error`.

```
####################
#
# submit description file
# Example 1: queuing multiple jobs with differing
# command line arguments and output files.
#
####################
```

```
Executable     = foo
Universe       = standard

Arguments      = 15 2000
Output  = foo.out1
Error   = foo.err1
Queue

Arguments      = 30 2000
Output  = foo.out2
Error   = foo.err2
Queue

Arguments      = 45 6000
Output  = foo.out3
Error   = foo.err3
Queue
```

- Submit Description File Example 2: This submit description file example queues 150 runs of program *foo* which must have been compiled and linked for an Intel x86 processor running RHEL 3. Condor will not attempt to run the processes on machines which have less than 32 Megabytes of physical memory, and it will run them on machines which have at least 64 Megabytes, if such machines are available. Stdin, stdout, and stderr will refer to `in.0`, `out.0`, and `err.0` for the first run of this program (process 0). Stdin, stdout, and stderr will refer to `in.1`, `out.1`, and `err.1` for process 1, and so forth. A log file containing entries about where and when Condor runs, takes checkpoints, and migrates processes in this cluster will be written into file `foo.log`.

```
####################
#
# Example 2: Show off some fancy features including
# use of pre-defined macros and logging.
#
####################

Executable     = foo
Universe       = standard
Requirements   = Memory >= 32 && OpSys == "LINUX" && Arch =="INTEL"
Rank           = Memory >= 64
Image_Size     = 28 Meg

Error   = err.$(Process)
Input   = in.$(Process)
Output  = out.$(Process)
Log = foo.log

Queue 150
```

- Command Line example: The following command uses the **-append** option to add two commands before the job(s) is queued. A log file and an error log file are specified. The submit description file is unchanged.

```
condor_submit -a "log = out.log" -a "error = error.log" mysubmitfile
```

Note that each of the added commands is contained within quote marks because there are space characters within the command.

- `periodic_remove` example: A job should be removed from the queue, if the total suspension time of the job is more than half of the run time of the job.
  Including the command

  ```
  periodic_remove = CumulativeSuspensionTime >
                    ((RemoteWallClockTime - CumulativeSuspensionTime) / 2.0)
  ```

  in the submit description file causes this to happen.

General Remarks

- For security reasons, Condor will refuse to run any jobs submitted by user root (UID = 0) or by a user whose default group is group wheel (GID = 0). Jobs submitted by user root or a user with a default group of wheel will appear to sit forever in the queue in an idle state.
- All path names specified in the submit description file must be less than 256 characters in length, and command line arguments must be less than 4096 characters in length; otherwise, *condor_submit* gives a warning message but the jobs will not execute properly.
- Somewhat understandably, behavior gets bizarre if the user makes the mistake of requesting multiple Condor jobs to write to the same file, and/or if the user alters any files that need to be accessed by a Condor job which is still in the queue. For example, the compressing of data or output files before a Condor job has completed is a common mistake.
- To disable checkpointing for Standard Universe jobs, include the line:

  ```
  +WantCheckpoint = False
  ```

  in the submit description file before the queue command(s).

See Also

Condor User Manual

Author

Condor Team, University of Wisconsin–Madison

Copyright

condor_submit_dag

Manage and queue jobs within a specified DAG for execution on remote machines

Synopsis

condor_submit_dag [**-help** | **-version**]

condor_submit_dag [**-no_submit**] [**-verbose**] [**-force**] [**-maxidle** *NumberOfJobs*] [**-maxjobs** *NumberOfJobs*] [**-dagman** *DagmanExecutable*] [**-maxpre** *NumberOfPREscripts*] [**-maxpost** *NumberOfPOSTscripts*] [**-notification** *value*] [**-noeventchecks**] [**-allowlogerror**] [**-r** *schedd_name*] [**-debug** *level*] [**-usedagdir**] [**-outfile_dir** *directory*] [**-config** *ConfigFileName*] [**-insert_sub_file** *FileName*] [**-append** *Command*] [**-oldrescue** *0|1*] [**-autorescue** *0|1*] [**-dorescuefrom** *number*] [**-allowversionmismatch**] [**-no_recurse**] [**-do_recurse**] [**-update_submit**] [**-import_env**] [**-DumpRescue**] [**-valgrind**] *DAGInputFile1* [*DAGInputFile2 . . .DAGInputFileN*]

Description

condor_submit_dag is the program for submitting a DAG (directed acyclic graph) of jobs for execution under Condor. The program enforces the job dependencies defined in one or more *DAGInputFile*s. Each *DAGInputFile* contains commands to direct the submission of jobs implied by the nodes of a DAG to Condor. See the Condor User Manual, section 2.10 for a complete description.

Options

-help Display usage information.

-version Display version information.

-no_submit Produce the Condor submit description file for DAGMan, but do not submit DAGMan as a Condor job.

-verbose Cause *condor_submit_dag* to give verbose error messages.

-force Require *condor_submit_dag* to overwrite the files that it produces, if the files already exist. Note that `dagman.out` will be appended to, not overwritten. If new-style rescue DAG mode is in effect, and any new-style rescue DAGs exist, the **-force** flag will cause them to be renamed, and the original DAG will be run. If old-style rescue DAG mode is in effect,

any existing old-style rescue DAGs will be deleted, and the original DAG will be run. Section 2.10.7 details rescue DAGs.

-maxidle ***NumberOfJobs*** Sets the maximum number of idle jobs allowed before *condor_dagman* stops submitting more jobs. Once idle jobs start to run, *condor_dagman* will resume submitting jobs. *NumberOfJobs* is a positive integer. If the option is omitted, the number of idle jobs is unlimited. Note that for this argument, each individual process within a cluster counts as a job, which is inconsistent with **-maxjobs** .

-maxjobs ***NumberOfJobs*** Sets the maximum number of jobs within the DAG that will be submitted to Condor at one time. *NumberOfJobs* is a positive integer. If the option is omitted, the default number of jobs is unlimited. Note that for this argument, each cluster counts as one job, no matter how many individual processes are in the cluster.

-dagman ***DagmanExecutable*** Allows the specification of an alternate *condor_dagman* executable to be used instead of the one found in the user's path. This must be a fully qualified path.

-maxpre ***NumberOfPREscripts*** Sets the maximum number of PRE scripts within the DAG that may be running at one time. *NumberOfPREScripts* is a positive integer. If this option is omitted, the default number of PRE scripts is unlimited.

-maxpost ***NumberOfPOSTscripts*** Sets the maximum number of POST scripts within the DAG that may be running at one time. *NumberOfPOSTScripts* is a positive integer. If this option is omitted, the default number of POST scripts is unlimited.

-notification ***value*** Sets the e-mail notification for DAGMan itself. This information will be used within the Condor submit description file for DAGMan. This file is produced by *condor_submit_dag*. See **notification** within the section of submit description file commands in the *condor_submit* manual page on page 825 for specification of *value.*

-noeventchecks This argument is no longer used; it is now ignored. Its functionality is now implemented by the `DAGMAN_ALLOW_EVENTS` configuration macro (see section 3.3.26).

-allowlogerror This optional argument has *condor_dagman* try to run the specified DAG, even in the case of detected errors in the user log specification. As of version 7.3.2, this argument has an effect only on DAGs containing Stork job nodes.

-r ***schedd_name*** Submit *condor_dagman* to a remote machine, specifically the *condor_schedd* daemon on that machine. The *condor_dagman* job will not run on the local *condor_schedd* (the submit machine), but on the specified one. This is implemented using the **-remote**

option to *condor_submit*. Note that this option does not currently specify input files for *condor_dagman*, nor the individual nodes to be taken along! It is assumed that any necessary files will be present on the remote computer, possibly via a shared file system between the local computer and the remote computer. It is also necessary that the user has appropriate permissions to submit a job to the remote machine; the permissions are the same as those required to use *condor_submit*'s **-remote** option. If other options are desired, including transfer of other input files, consider using the **-no_submit** option, modifying the resulting submit file for specific needs, and then using *condor_submit* on that.

-debug *level* Passes the the *level* of debugging output desired to *condor_dagman*. *level* is an integer, with values of 0-7 inclusive, where 7 is the most verbose output. A default value of 3 is passed to *condor_dagman* when not specified with this option. See the *condor_dagman* manual page on page 729 for detailed descriptions of these values.

-usedagdir This optional argument causes *condor_dagman* to run each specified DAG as if *condor_submit_dag* had been run in the directory containing that DAG file. This option is most useful when running multiple DAGs in a single *condor_dagman*. Note that the **-usedagdir** flag must not be used when running an old-style rescue DAG (see section 2.10.7).

-outfile_dir *directory* Specifies the directory in which the `.dagman.out` file will be written. The *directory* may be specified relative to the current working directory as *condor_submit_dag* is executed, or specified with an absolute path. Without this option, the `.dagman.out` file is placed in the same directory as the first DAG input file listed on the command line.

-config *ConfigFileName* Specifies a configuration file to be used for this DAGMan run. Note that the options specified in the configuration file apply to all DAGs if multiple DAGs are specified. Further note that it is a fatal error if the configuration file specified by this option conflicts with a configuration file specified in any of the DAG files, if they specify one. For more information about how *condor_dagman* configuration files work, see section 2.10.6.

-insert_sub_file *FileName* Specifies a file to insert into the `.condor.sub` file created by *condor_submit_dag*. The specified file must contain only legal submit file commands. Only one file can be inserted. (If both the DAGMAN_INSERT_SUB_FILE configuration variable and **-insert_sub_file** are specified, **-insert_sub_file** overrides DAGMAN_INSERT_SUB_FILE.) The specified file is inserted into the `.condor.sub` file before the Queue command and before any commands specified with the **-append** option.

-append *Command* Specifies a command to append to the `.condor.sub` file created by *condor_submit_dag*. The specified command is appended to the `.condor.sub` file immediately before the Queue command. Multiple commands are specified by using the **-append** option multiple times. Each new command is given in a separate **-append** option. Commands with spaces in them must be enclosed in double quotes. Commands specified

with the **-append** option are appended to the `.condor.sub` file *after* commands inserted from a file specified by the **-insert_sub_file** option or the DAGMAN_INSERT_SUB_FILE configuration variable, so the **-append** command(s) will override commands from the inserted file if the commands conflict.

-oldrescue *0|1* Whether to use "old-style" rescue DAG naming (see section 2.10.7) when creating a rescue DAG (0 = `false`, 1 = `true`).

-autorescue *0|1* Whether to automatically run the newest rescue DAG for the given DAG file, if one exists (0 = `false`, 1 = `true`).

-dorescuefrom *number* Forces *condor_dagman* to run the specified rescue DAG number for the given DAG. A value of 0 is the same as not specifying this option. Specifying a non-existent rescue DAG is a fatal error.

-allowversionmismatch This optional argument causes *condor_dagman* to allow a version mismatch between *condor_dagman* itself and the `.condor.sub` file produced by *condor_submit_dag* (or, in other words, between *condor_submit_dag* and *condor_dagman*). WARNING! This option should be used only if absolutely necessary. Allowing version mismatches can cause subtle problems when running DAGs. (Note that, starting with version 7.4.0, *condor_dagman* no longer requires an exact version match between itself and the `.condor.sub` file. Instead, a "minimum compatible version" is defined, and any `.condor.sub` file of that version or newer is accepted.)

-no_recurse This optional argument causes *condor_submit_dag* to *not* run itself recursively on nested DAGs (this is now the default; this flag has been kept mainly for backwards compatibility).

-do_recurse This optional argument causes *condor_submit_dag* to run itself recursively on nested DAGs (the default is now that it does *not* run itself recursively; instead the `.condor.sub` files for nested DAGs are generated "lazily" by *condor_dagman* itself). (DAG nodes specified with the `SUBDAG EXTERNAL` keyword or with submit file names ending in `.condor.sub` are considered nested DAGs.) (See also the `DAGMAN_GENERATE_SUBDAG_SUBMITS` configuration variable in section 3.3.26 for more information.)

-update_submit This optional argument causes an existing `.condor.sub` file to not be treated as an error; rather, the `.condor.sub` file will be overwritten, but the existing values of **-maxjobs**, **-maxidle**, **-maxpre**, and **-maxpost** will be preserved.

-import_env This optional argument causes *condor_submit_dag* to import the current environment into the **environment** command of the `.condor.sub` file it generates.

-DumpRescue This optional argument tells *condor_dagman* to immediately dump a rescue DAG and then exit, as opposed to actually running the DAG. This feature is mainly intended for testing. The Rescue DAG file is produced whether or not there are parse errors reading the original DAG input file. The name of the file differs if there was a parse error.

-valgrind This optional argument causes the submit description file generated for the submission of *condor_dagman* to be modified. The executable becomes *valgrind* run on *condor_dagman*, with a specific set of arguments intended for testing *condor_dagman*. Note that this argument is intended for testing purposes only. Using the **-valgrind** option without the necessary *valgrind* software installed will cause the DAG to fail. If the DAG does run, it will run much more slowly than usual.

See Also

Condor User Manual

Exit Status

condor_submit_dag will exit with a status value of 0 (zero) upon success, and it will exit with the value 1 (one) upon failure.

Examples

To run a single DAG:

```
% condor_submit_dag diamond.dag
```

To run a DAG when it has already been run and the output files exist:

```
% condor_submit_dag -force diamond.dag
```

To run a DAG, limiting the number of idle node jobs in the DAG to a maximum of five:

```
% condor_submit_dag -maxidle 5 diamond.dag
```

To run a DAG, limiting the number of concurrent PRE scripts to 10 and the number of concurrent POST scripts to five:

```
% condor_submit_dag -maxpre 10 -maxpost 5 diamond.dag
```

To run two DAGs, each of which is set up to run in its own directory:

```
% condor_submit_dag -usedagdir dag1/diamond1.dag dag2/diamond2.dag
```

Author

Condor Team, University of Wisconsin–Madison

Copyright

condor_transfer_data

transfer spooled data

Synopsis

condor_transfer_data [**-help** | **-version**]

condor_transfer_data [**-pool** *centralmanagerhostname[:portnumber]* | **-name** *scheddname*]| [**-addr** *"<a.b.c.d:port>"*] *cluster. . .*| *cluster.process. . .*| *user. . .* | **-constraint** *expression . . .*

condor_transfer_data [**-pool** *centralmanagerhostname[:portnumber]* | **-name** *scheddname*]| [**-addr** *"<a.b.c.d:port>"*] **-all**

Description

condor_transfer_data causes Condor to transfer spooled data. It is meant to be used in conjunction with the **-spool** option of *condor_submit*, as in

```
condor_submit -spool mysubmitfile
```

Submission of a job with the **-spool** option causes Condor to spool all input files, the user log, and any proxy across a connection to the machine where the *condor_schedd* daemon is running. After spooling these files, the machine from which the job is submitted may disconnect from the network or modify its local copies of the spooled files.

When the job finishes, the job has `JobStatus` = 4, meaning that the job has completed. The output of the job is spooled, and *condor_transfer_data* retrieves the output of the completed job.

Options

-help Display usage information

-version Display version information

-pool *centralmanagerhostname[:portnumber]* Specify a pool by giving the central manager's host name and an optional port number

-name *scheddname* Send the command to a machine identified by *scheddname*

-addr *"<**a.b.c.d:port**>"* Send the command to a machine located at *"<a.b.c.d:port>"*

cluster Transfer spooled data belonging to the specified cluster

cluster.process Transfer spooled data belonging to a specific job in the cluster

user Transfer spooled data belonging to the specified user

-constraint ***expression*** Transfer spooled data for jobs which match the job ClassAd expression constraint

-all Transfer all spooled data

Exit Status

condor_transfer_data will exit with a status value of 0 (zero) upon success, and it will exit with the value 1 (one) upon failure.

Author

Condor Team, University of Wisconsin–Madison

Copyright

condor_updates_stats

Display output from *condor_status*

Synopsis

condor_updates_stats [**--help** | **-h**] | [**--version**]

condor_updates_stats [**--long** | **-l**] [**--history=**<**min**>**-**<**max**>] [**--interval=**<**seconds**>] [**--notime**] [**--time**] [**--summary** | **-s**]

Description

condor_updates_stats parses the output from *condor_status*, and it displays the information relating to update statistics in a useful format. The statistics are displayed with the most recent update first; the most recent update is numbered with the smallest value.

The number of historic points that represent updates is configurable on a per-source basis. See `COLLECTOR_DAEMON_HISTORY_SIZE` in section 3.3.16.

Options

—help Display usage information and exit.

-h Same as **—help**.

—version Display Condor version information and exit.

—long All update statistics are displayed. Without this option, the statistics are condensed.

-l Same as **—long**.

—history=<**min**>**-**<**max**> Sets the range of update numbers that are printed. By default, the entire history is displayed. To limit the range, the minimum and/or maximum number may be specified. If a minimum is not specified, values from 0 to the maximum are displayed. If the maximum is not specified, all values after the minimum are displayed. When both minimum and maximum are specified, the range to be displayed includes the endpoints as well as all values in between. If no = sign is given, command-line parsing fails, and usage information is displayed. If an = sign is given, with no minimum or maximum values, the default of the

entire history is displayed.

—interval=<**seconds**> The assumed update interval, in seconds. Assumed times for the the updates are displayed, making the use of the **—time** option together with the **—interval** option redundant.

—notime Do not display assumed times for the the updates. If more than one of the options **—notime** and **—time** are provided, the final one within the command line parsed determines the display.

—time Display assumed times for the the updates. If more than one of the options **—notime** and **—time** are provided, the final one within the command line parsed determines the display.

—summary Display only summary information, not the entire history for each machine.

-s Same as **—summary**.

Exit Status

condor_updates_stats will exit with a status value of 0 (zero) upon success, and it will exit with a nonzero value upon failure.

Examples

Assuming the default of 128 updates kept, and assuming that the update interval is 5 minutes, *condor_updates_stats* displays:

```
$ condor_status -l host1 | condor_updates_stats --interval=300
(Reading from stdin)
*** Name/Machine = 'HOST1.cs.wisc.edu' MyType = 'Machine' ***
 Type: Main
   Stats: Total=2277, Seq=2276, Lost=3 (0.13%)
     0 @ Mon Feb 16 12:55:38 2004: Ok
  ...
    28 @ Mon Feb 16 10:35:38 2004: Missed
    29 @ Mon Feb 16 10:30:38 2004: Ok
  ...
   127 @ Mon Feb 16 02:20:38 2004: Ok
```

Within this display, update numbered 27, which occurs later in time than the missed update numbered 28, is Ok. Each change in state, in reverse time order, displays in this condensed version.

Author

Condor Team, University of Wisconsin–Madison

Copyright

condor_userlog

Display and summarize job statistics from job log files.

Synopsis

condor_userlog [**-help**] [**-total** | **-raw**] [**-debug**] [**-evict**] [**-j** *cluster* | *cluster.proc*] [**-all**] [**-hostname**] *logfile . . .*

Description

condor_userlog parses the information in job log files and displays summaries for each workstation allocation and for each job. See the manual page for *condor_submit* on page 825 for instructions for specifying that Condor write a log file for your jobs.

If **-total** is not specified, *condor_userlog* will first display a record for each workstation allocation, which includes the following information:

Job The cluster/process id of the Condor job.

Host The host where the job ran. By default, the host's IP address is displayed. If **-hostname** is specified, the host name will be displayed instead.

Start Time The time (month/day hour:minute) when the job began running on the host.

Evict Time The time (month/day hour:minute) when the job was evicted from the host.

Wall Time The time (days+hours:minutes) for which this workstation was allocated to the job.

Good Time The allocated time (days+hours:min) which contributed to the completion of this job. If the job exited during the allocation, then this value will equal "Wall Time." If the job performed a checkpoint, then the value equals the work saved in the checkpoint during this allocation. If the job did not exit or perform a checkpoint during this allocation, the value will be 0+00:00. This value can be greater than 0 and less than "Wall Time" if the application completed a periodic checkpoint during the allocation but failed to checkpoint when evicted.

CPU Usage The CPU time (days+hours:min) which contributed to the completion of this job.

condor_userlog will then display summary statistics per host:

Host/Job The IP address or host name for the host.

Wall Time The workstation time (days+hours:minutes) allocated by this host to the jobs specified in the query. By default, all jobs in the log are included in the query.

Good Time The time (days+hours:minutes) allocated on this host which contributed to the completion of the jobs specified in the query.

CPU Usage The CPU time (days+hours:minutes) obtained from this host which contributed to the completion of the jobs specified in the query.

Avg Alloc The average length of an allocation on this host (days+hours:minutes).

Avg Lost The average amount of work lost (days+hours:minutes) when a job was evicted from this host without successfully performing a checkpoint.

Goodput This percentage is computed as Good Time divided by Wall Time.

Util. This percentage is computed as CPU Usage divided by Good Time.

condor_userlog will then display summary statistics per job:

Host/Job The cluster/process id of the Condor job.

Wall Time The total workstation time (days+hours:minutes) allocated to this job.

Good Time The total time (days+hours:minutes) allocated to this job which contributed to the job's completion.

CPU Usage The total CPU time (days+hours:minutes) which contributed to this job's completion.

Avg Alloc The average length of a workstation allocation obtained by this job in minutes (days+hours:minutes).

Avg Lost The average amount of work lost (days+hours:minutes) when this job was evicted from a host without successfully performing a checkpoint.

Goodput This percentage is computed as Good Time divided by Wall Time.

Util. This percentage is computed as CPU Usage divided by Good Time.

Finally, *condor_userlog* will display a summary for all hosts and jobs.

Options

-help Get a brief description of the supported options

-total Only display job totals

-raw Display raw data only

-debug Debug mode

-j Select a specific cluster or cluster.proc

-evict Select only allocations which ended due to eviction

-all Select all clusters and all allocations

-hostname Display host name instead of IP address

General Remarks

Since the Condor job log file format does not contain a year field in the timestamp, all entries are assumed to occur in the current year. Allocations which begin in one year and end in the next will be silently ignored.

Exit Status

condor_userlog will exit with a status value of 0 (zero) upon success, and it will exit with the value 1 (one) upon failure.

Author

Condor Team, University of Wisconsin–Madison

Copyright

condor_userprio

Manage user priorities

Synopsis

condor_userprio [**-pool** *centralmanagerhostname[:portnumber]*] [**-all**] [**-usage**] [**-setprio** *username value*] [**-setfactor** *username value*] [**-setaccum** *username value*] [**-setbegin** *username value*] [**-setlast** *username value*] [**-resetusage** *username*] [**-resetall**] [**-delete** *username*] [**-getreslist** *username*] [**-allusers**] [**-activefrom** *month day year*] [**-l**]

Description

condor_userprio with no arguments, lists the active users (see below) along with their priorities, in increasing priority order. The -all option can be used to display more detailed information about each user, which includes the following columns:

Effective Priority The effective priority value of the user, which is used to calculate the user's share when allocating resources. A lower value means a higher priority, and the minimum value (highest priority) is 0.5. The effective priority is calculated by multiplying the real priority by the priority factor.

Real Priority The value of the real priority of the user. This value follows the user's resource usage.

Priority Factor The system administrator can set this value for each user, thus controlling a user's effective priority relative to other users. This can be used to create different classes of users.

Res Used The number of resources currently used (e.g. the number of running jobs for that user).

Accumulated Usage The accumulated number of resource-hours used by the user since the usage start time.

Usage Start Time The time since when usage has been recorded for the user. This time is set when a user job runs for the first time. It is reset to the present time when the usage for the user is reset (with the -resetusage or -resetall options).

Last Usage Time The most recent time a resource usage has been recorded for the user.

The -usage option displays the username, accumulated usage, usage start time and last usage time for each user, sorted on accumulated usage.

The -setprio, -setfactor options are used to change a user's real priority and priority factor. The -setaccum option sets a user's accumulated usage. The -setbegin, -setlast options are used to change a user's begin usage time and last usage time. The -setaccum option sets a user's accumulated usage.

The -resetusage and -resetall options are used to reset the accumulated usage for users. The usage start time is set to the current time when the accumulated usage is reset. These options require administrator privileges.

By default only users for whom usage was recorded in the last 24 hours or whose priority is greater than the minimum are listed. The -activefrom and -allusers options can be used to display users who had some usage since a specified date, or ever. The summary line for last usage time will show this date.

The -getreslist option is used to display the resources currently used by a user. The output includes the start time (the time the resource was allocated to the user), and the match time (how long has the resource been allocated to the user).

Note that when specifying user names on the command line, the name must include the UID domain (e.g. user@uid-domain - exactly the same way user names are listed by the userprio command).

The -pool option can be used to contact a different central-manager instead of the local one (the default).

For security purposes (authentication and authorization), this command requires an administrator's level of access. See section 3.6.1 on page 315 for further explanation.

Options

-pool *centralmanagerhostname[:portnumber]* Contact specified *centralmanagerhostname* with an optional port number instead of the local central manager. This can be used to check other pools. NOTE: The host name (and optionally port) specified refer to the host name (and port) of the *condor_negotiator* to query for user priorities. This is slightly different than most Condor tools that support -pool, which expect the host name (and optionally port) of the *condor_collector*, instead.

-all Display detailed information about each user.

-usage Display usage information for each user.

-setprio *username value* Set the real priority of the specified user to the specified value.

-setfactor *username value* Set the priority factor of the specified user to the specified value.

-setaccum *username value* Set the accumulated usage of the specified user to the specified floating point value.

-setbegin *username value* Set the begin usage time of the specified user to the specified value.

-setlast *username value* Set the last usage time of the specified user to the specified value.

-resetusage *username* Reset the accumulated usage of the specified user to zero.

-resetall Reset the accumulated usage of all the users to zero.

-delete *username* Remove the specified *username* from Condor's accounting.

-getreslist *username* Display all the resources currently allocated to the specified user.

-allusers Display information for all the users who have some recorded accumulated usage.

-activefrom *month day year* Display information for users who have some recorded accumulated usage since the specified date.

-l Show the class-ad which was received from the central-manager in long format.

Exit Status

condor_userprio will exit with a status value of 0 (zero) upon success, and it will exit with the value 1 (one) upon failure.

Author

Condor Team, University of Wisconsin–Madison

Copyright

condor_vacate

Vacate jobs that are running on the specified hosts

Synopsis

condor_vacate [**-help** | **-version**]

condor_vacate [**-graceful** | **-fast**] [**-debug**] [**-pool** *centralmanagerhostname[:portnumber]*] [**-name** *hostname* | *hostname* | **-addr** *"<a.b.c.d:port>"* | *"<a.b.c.d:port>"* | **-constraint** *expression* | **-all**]

Description

condor_vacate causes Condor to checkpoint any running jobs on a set of machines and force the jobs to vacate the machine. The job(s) remains in the submitting machine's job queue.

Given the (default) **-graceful** option, a job running under the standard universe will first produce a checkpoint and then the job will be killed. Condor will then restart the job somewhere else, using the checkpoint to continue from where it left off. A job running under the vanilla universe is killed, and Condor restarts the job from the beginning somewhere else. *condor_vacate* has no effect on a machine with no Condor job currently running.

There is generally no need for the user or administrator to explicitly run *condor_vacate*. Condor takes care of jobs in this way automatically following the policies given in configuration files.

Options

-help Display usage information

-version Display version information

-graceful Inform the job to checkpoint, then soft-kill it.

-fast Hard-kill jobs instead of checkpointing them

-debug Causes debugging information to be sent to `stderr`, based on the value of the configuration variable `TOOL_DEBUG`

-pool ***centralmanagerhostname[:portnumber]*** Specify a pool by giving the central manager's host name and an optional port number

-name ***hostname*** Send the command to a machine identified by *hostname*

hostname Send the command to a machine identified by *hostname*

-addr *"<**a.b.c.d:port**>"* Send the command to a machine's master located at *"<a.b.c.d:port>"*

*"<**a.b.c.d:port**>"* Send the command to a machine located at *"<a.b.c.d:port>"*

-constraint ***expression*** Apply this command only to machines matching the given ClassAd *expression*

-all Send the command to all machines in the pool

Exit Status

condor_vacate will exit with a status value of 0 (zero) upon success, and it will exit with the value 1 (one) upon failure.

Examples

To send a *condor_vacate* command to two named machines:

```
% condor_vacate  robin cardinal
```

To send the *condor_vacate* command to a machine within a pool of machines other than the local pool, use the **-pool** option. The argument is the name of the central manager for the pool. Note that one or more machines within the pool must be specified as the targets for the command. This command sends the command to a the single machine named **cae17** within the pool of machines that has **condor.cae.wisc.edu** as its central manager:

```
% condor_vacate -pool condor.cae.wisc.edu -name cae17
```

Author

Condor Team, University of Wisconsin–Madison

Copyright

condor_vacate_job

vacate jobs in the Condor queue from the hosts where they are running

Synopsis

condor_vacate_job [**-help** | **-version**]

condor_vacate_job [**-pool** *centralmanagerhostname[:portnumber]* | **-name** *scheddname*]| [**-addr** *"<a.b.c.d:port>"*] [**-fast**] *cluster...*| *cluster.process...*| *user...* | **-constraint** *expression ...*

condor_vacate_job [**-pool** *centralmanagerhostname[:portnumber]* | **-name** *scheddname*]| [**-addr** *"<a.b.c.d:port>"*] [**-fast**] **-all**

Description

condor_vacate_job finds one or more jobs from the Condor job queue and vacates them from the host(s) where they are currently running. The jobs remain in the job queue and return to the idle state.

A job running under the standard universe will first produce a checkpoint and then the job will be killed. Condor will then restart the job somewhere else, using the checkpoint to continue from where it left off. A job running under any other universe will be sent a soft kill signal (SIGTERM by default, or whatever is defined as the `SoftKillSig` in the job ClassAd), and Condor will restart the job from the beginning somewhere else.

If the **-fast** option is used, the job(s) will be immediately killed, meaning that standard universe jobs will not be allowed to checkpoint, and the job will have to revert to the last checkpoint or start over from the beginning.

If the **-name** option is specified, the named *condor_schedd* is targeted for processing. If the **-addr** option is used, the *condor_schedd* at the given address is targeted for processing. Otherwise, the local *condor_schedd* is targeted. The jobs to be vacated are identified by one or more job identifiers, as described below. For any given job, only the owner of the job or one of the queue super users (defined by the `QUEUE_SUPER_USERS` macro) can vacate the job.

Using *condor_vacate_job* on jobs which are not currently running has no effect.

Options

-help Display usage information

-version Display version information

-pool *centralmanagerhostname[:portnumber]* Specify a pool by giving the central manager's host name and an optional port number

-name *scheddname* Send the command to a machine identified by *scheddname*

-addr *"<**a.b.c.d:port**>"* Send the command to a machine located at *"<a.b.c.d:port>"*

cluster Vacate all jobs in the specified cluster

cluster.process Vacate the specific job in the cluster

user Vacate jobs belonging to specified user

-constraint *expression* Vacate all jobs which match the job ClassAd expression constraint

-all Vacate all the jobs in the queue

-fast Perform a fast vacate and hard kill the jobs

General Remarks

Do not confuse *condor_vacate_job* with *condor_vacate*. *condor_vacate* is given a list of hosts to vacate, regardless of what jobs happen to be running on them. Only machine owners and administrators have permission to use *condor_vacate* to evict jobs from a given host. *condor_vacate_job* is given a list of job to vacate, regardless of which hosts they happen to be running on. Only the owner of the jobs or queue super users have permission to use *condor_vacate_job*.

Examples

To vacate job 23.0:

```
% condor_vacate_job 23.0
```

To vacate all jobs of a user named Mary:

```
% condor_vacate_job mary
```

To vacate all standard universe jobs owned by Mary:

```
% condor_vacate_job -constraint 'JobUniverse == 1 && Owner == "mary"'
```

Note that the entire constraint, including the quotation marks, must be enclosed in single quote marks for most shells.

Exit Status

condor_vacate_job will exit with a status value of 0 (zero) upon success, and it will exit with the value 1 (one) upon failure.

Author

Condor Team, University of Wisconsin–Madison

Copyright

condor_version

print Condor version and platform information

Synopsis

condor_version [**-help**]

condor_version [**-arch**] [**-opsys**] [**-syscall**]

Description

With no arguments, *condor_version* prints the currently installed Condor version number and platform information. The version number includes a build identification number, as well as the date built.

Options

help Print usage information

arch Print this machine's ClassAd value for `Arch`

opsys Print this machine's ClassAd value for `OpSys`

syscall Get any requested version and/or platform information from the `libcondorsyscall.a` that this Condor pool is configured to use, instead of using the values that are compiled into the tool itself. This option may be used in combination with any other options to modify where the information is coming from.

Exit Status

condor_version will exit with a status value of 0 (zero) upon success, and it should never exit with a failing value.

Author

Condor Team, University of Wisconsin–Madison

Copyright

condor_wait

Wait for jobs to finish

Synopsis

condor_wait [**-help** | **-version**]

condor_wait [**-debug**] [**-wait** *seconds*] [**-num** *number-of-jobs*] *log-file* [**job ID**]

Description

condor_wait watches a user log file (created with the **log** command within a submit description file) and returns when one or more jobs from the log have completed or aborted.

Because *condor_wait* expects to find at least one job submitted event in the log file, at least one job must have been successfully submitted with *condor_submit* before *condor_wait* is executed.

condor_wait will wait forever for jobs to finish, unless a shorter wait time is specified.

Options

-help Display usage information

-version Display version information

-debug Show extra debugging information.

-wait *seconds* Wait no more than the integer number of *seconds.* The default is unlimited time.

-num *number-of-jobs* Wait for the integer *number-of-jobs* jobs to end. The default is all jobs in the log file.

log file The name of the log file to watch for information about the job.

job ID A specific job or set of jobs to watch. If the **job ID** is only the job ClassAd attribute `ClusterId`, then *condor_wait* waits for all jobs with the given `ClusterId`. If the **job ID** is a pair of the job ClassAd attributes, given by `ClusterId`.`ProcId`, then *condor_wait* waits for the specific job with this **job ID**. If this option is not specified, all jobs that exist in

the log file when *condor_wait* is invoked will be watched.

General Remarks

condor_wait is an inexpensive way to test or wait for the completion of a job or a whole cluster, if you are trying to get a process outside of Condor to synchronize with a job or set of jobs.

It can also be used to wait for the completion of a limited subset of jobs, via the **-num** option.

Examples

```
condor_wait logfile
```

This command waits for all jobs that exist in `logfile` to complete.

```
condor_wait logfile 40
```

This command waits for all jobs that exist in `logfile` with a job ClassAd attribute `ClusterId` of 40 to complete.

```
condor_wait -num 2 logfile
```

This command waits for any two jobs that exist in `logfile` to complete.

```
condor_wait logfile 40.1
```

This command waits for job 40.1 that exists in `logfile` to complete.

```
condor_wait -wait 3600 logfile 40.1
```

This waits for job 40.1 to complete by watching `logfile`, but it will not wait more than one hour (3600 seconds).

Exit Status

condor_wait exits with 0 if and only if the specified job or jobs have completed or aborted. *condor_wait* returns 1 if unrecoverable errors occur, such as a missing log file, if the job does not exist in the log file, or the user-specified waiting time has expired.

Author

Condor Team, University of Wisconsin–Madison

Copyright

filelock_midwife

create an artifact of the creation of a process

Synopsis

***filelock_midwife* -help**

filelock_midwife [**–file** *filename*] *program* [*programargs*]

Description

filelock_midwife starts a given *program*, while creating an artifact of the program's birth. At a later time the *filelock_undertaker* can examine the artifact to determine whether the program is still running, or whether the program has exited. *filelock_midwife* accomplishes this by obtaining a file lock on the given artifact file before starting the program.

Warning: *filelock_midwife* will not work on NFS unless the separate file lock server is running.

Options

–file *filename* The *filename* to use for the artifact file. The file `lock.file` is the default file used when this option is not specified.

***program* [*programargs*]** Forks a process and executes *program* with *programargs* as command-line arguments (when specified).

Exit Status

filelock_midwife will exit with a status of 0 (zero) upon success, and non-zero otherwise.

See Also

uniq_pid_midwife (on page 897), *filelock_undertaker* (on page 889).

Author

Condor Team, University of Wisconsin–Madison

Copyright

filelock_undertaker

determine whether a process has exited

Synopsis

***filelock_undertaker* -help**

filelock_undertaker [**–file** *filename*] [**–block**]

Description

filelock_undertaker can examine an artifact file created by *filelock_midwife* and determine whether the program started by the *midwife* has exited. It does this by attempting to acquire a file lock.

Be warned that this will not work on NFS unless the separate file lock server is running.

Options

–block If the process has not exited, block until it does.

–file *filename* The name of the artifact file. created by *filelock_midwife*. The file `lock.file` is the default file used when this option is not specified.

Exit Status

filelock_undertaker will exit with a status of 0 (zero) if the monitored process has exited, with a status of 1 (one) if the monitored process has definitely not exited, with a status of 2 if it is uncertain whether the process has exited (this is generally due to a failure by the *filelock_midwife*), or with any other value for program failure.

See Also

uniq_pid_undertaker (on page 899), *filelock_midwife* (on page 887).

Author

Condor Team, University of Wisconsin–Madison

Copyright

gidd_alloc

find a GID within the specified range which is not used by any process

Synopsis

gidd_alloc *min-gid max-gid*

Description

This program will scan the alive PIDs, looking for which GID is unused in the supplied, inclusive range specified by the required arguments *min-gid* and *max-gid*. Upon finding one, it will add the GID to its own supplementary group list, and then scan the PIDs again expecting to find only itself using the GID. If no collision has occurred, the program exits, otherwise it retries.

General Remarks

This is a program only available for the Linux ports of Condor.

Exit Status

gidd_alloc will exit with a status value of 0 (zero) upon success, and it will exit with the value 1 (one) upon failure.

Author

Condor Team, University of Wisconsin–Madison

Copyright

install_release

install an arbitrary software release into a named directory

Synopsis

install_release [**-help**]

install_release [**-f**] [**-basedir** *directory*] [**-log** *filename*] [**-wget**] [**-globuslocation** *directory*] [**-o** *otherfile1...*] *package*

Description

install_release installs an arbitrary software release into a named directory. In addition it creates a log of the installed files for easy uninstallation. This program can install packages of type tar, gzip, or gzip'ed tar. The installation package can be located on a mounted file system, an http server, an ftp server, or a grid ftp server.

Options

-basedir *directory* The directory where the package should be installed. When not specified, the directory defaults to the current working directory.

-f Forcefully overwrite files if they exist.

-globuslocation *directory* This program does not come prepackaged with *globus-url-copy* or the supporting libraries. If globus is not installed in the `/opt/globus` directory, the user must specify the installation location of globus using this option.

-help Display brief usage information and exit.

-log *filename* The file name for the installation log.

-o *otherfile1...* A space-separated list of files that will be installed along with the installation package. The files will only be copied. No extraction or decompression will be performed on these files. These files will be logged in the installation log.

package The full path to the installation package. Locations on file systems can be specified without the `file:` prefix, but other locations must prefix with the appropriate protocol (`http:`, `ftp:`, or `gsiftp:`).

-wget This program defaults to using *globus-url-copy* to fetch the installation package. This option specifies that this program should use *wget* for http and ftp requests and Perl's copy function for file system requests. *wget* must be installed on the machine and must be in the user's path.

Exit Status

install_release will exit with a status value of 0 (zero) upon success, and non-zero otherwise.

See Also

cleanup_release (on page 691)

Author

Condor Team, University of Wisconsin–Madison

Copyright

procd_ctl

command line interface to the *condor_procd*

Synopsis

procd_ctl -h

procd_ctl -A *address-file* [**command**]

Description

This is a programmatic interface to the *condor_procd* daemon. It may be used to cause the *condor_procd* to do anything that the *condor_procd* is capable of doing, such as tracking and managing process families.

This is a program only available for the Linux ports of Condor.

procd_ctl honors the discovery algorithm for Condor's configuration files, which specify debugging information. As such, if Condor is not installed or the configuration files are unavailable, then set the environment variable `CONDOR_CONFIG` to `/dev/null` to be utilized by the discovery algorithm.

The **-h** option prints out usage information and exits. The *address-file* specification within the **-A** argument specifies the path and file name of the address file which the named pipe clients must use to speak with the *condor_procd*.

One command is given to the *condor_procd*. The choices for the command are defined by the Options.

Options

TRACK_BY_ASSOCIATED_GID *GID* [*PID*] Use the specified *GID* to track the specified family rooted at *PID*. If the optional *PID* is not specified, then the PID used is the one given or assumed by *condor_procd*.

GET_USAGE [*PID*] Get the total usage information about the PID family at *PID*. If the optional *PID* is not specified, then the PID used is the one given or assumed by *condor_procd*.

DUMP [*PID*] Print out information about both the root *PID* being watched and the tree of processes under this root *PID*. If the optional *PID* is not specified, then the PID used is the one given or assumed by *condor_procd*.

LIST [*PID*] With no *PID* given, print out information about all the watched processes. If the optional *PID* is specified, print out information about the process specified by *PID* and all its child processes.

SIGNAL_PROCESS *signal* [*PID*] Send the *signal* to the process specified by *PID*. If the optional *PID* is not specified, then the PID used is the one given or assumed by *condor_procd*.

SUSPEND_FAMILY *PID* Suspend the process family rooted at *PID*.

CONTINUE_FAMILY *PID* Continue execution of the process family rooted at *PID*.

KILL_FAMILY *PID* Kill the process family rooted at *PID*.

UNREGISTER_FAMILY *PID* Stop tracking the process family rooted at *PID*.

SNAPSHOT Perform a snapshot of the tracked family tree.

QUIT Disconnect from the *condor_procd* and exit.

General Remarks

This program may be used in a standalone mode, independent of Condor, to track process families. The programs *procd_ctl* and *gidd_alloc* are used with the *condor_procd* in standalone mode to interact with the daemon and inquire about certain state of running processes on the machine, respectively.

Exit Status

procd_ctl will exit with a status value of 0 (zero) upon success, and it will exit with the value 1 (one) upon failure.

Author

Condor Team, University of Wisconsin–Madison

Copyright

uniq_pid_midwife

create an artifact of the creation of a process

Synopsis

uniq_pid_midwife [**--noblock**] [**--file** *filename*] [**--precision** *seconds*] *program* [*programargs*]

Description

uniq_pid_midwife starts a given program, while creating an artifact of the program's birth. At a later time the *uniq_pid_undertaker* can examine the artifact to determine whether the program is still running or whether it has exited. *uniq_pid_midwife* accomplishes this by recording an enforced unique process identifier to the artifact.

Options

--file *filename* The *filename* to use for the artifact file. Defaults to `pid.file`.

--precision *seconds* The precision the operating system is expected to have in regards to process creation times. Defaults to an operating system specific value. The default is the best choice in most cases.

--noblock Exit after the program has been confirmed, typically 3 times the precision. Defaults to block until the program exits.

***program* [*programargs*]** Forks a process and executes *program* with *programargs* as command-line arguments (when specified).

Exit Status

uniq_pid_midwife will exit with a status of 0 (zero) upon success, and non-zero otherwise.

See Also

uniq_pid_undertaker (on page 899), *filelock_midwife* (on page 887).

Author

Condor Team, University of Wisconsin–Madison

Copyright

uniq_pid_undertaker

determine whether a process has exited

Synopsis

uniq_pid_undertaker [**--block**] [**--file** *file*] [**--precision** *seconds*]

Description

uniq_pid_undertaker can examine an artifact file created by *uniq_pid_midwife* and determine whether the program started by the *midwife* has exited.

Options

--block If the process has not exited, block until it does.

--file *file* The name of the *uniq_pid_midwife* created artifact file. Defaults to `pid.file`.

--precision *seconds* Uses *seconds* as the precision range within which the operating system will provide a process's birthday. Defaults to an operating system specific value. Only use this option if the same *seconds* value was provided to *uniq_pid_midwife*.

Exit Status

uniq_pid_undertaker will exit with a status of 0 (zero) if the monitored process has exited, with a status of 1 (one) if the monitored process has definitely not exited, with a status of 2 if it is uncertain whether the process has exited (this is generally due to a failure by the *uniq_pid_midwife*), or with any other value for program failure.

See Also

uniq_pid_midwife (on page 897), *filelock_undertaker* (on page 889).

Author

Condor Team, University of Wisconsin–Madison

Copyright

CHAPTER

TEN

Appendix A: ClassAd Attributes

ClassAd Types

ClassAd attributes vary, depending on the entity producing the ClassAd. Therefore, each ClassAd has an attribute named `MyType`, which describes the type of ClassAd. In addition, the *condor_collector* appends attributes to any daemon's ClassAd, whenever the *condor_collector* is queried. These additional attributes are listed in the unnumbered subsection labeled ClassAd Attributes Added by the *condor_collector* on page 925.

Here is a list of defined values for `MyType`, as well as a reference to a list attributes relevant to that type.

`Job` Each submitted job describes its state, for use by the *condor_negotiator* daemon in finding a machine upon which to run the job. ClassAd attributes that appear in a job ClassAd are listed and described in the unnumbered subsection labeled Job ClassAd Attributes on page 902.

`Machine` Each machine in the pool (and hence, the *condor_startd* daemon running on that machine) describes its state. ClassAd attributes that appear in a machine ClassAd are listed and described in the unnumbered subsection labeled Machine ClassAd Attributes on page 913.

`DaemonMaster` Each *condor_master* daemon describes its state. ClassAd attributes that appear in a DaemonMaster ClassAd are listed and described in the unnumbered subsection labeled DaemonMaster ClassAd Attributes on page 920.

`Scheduler` Each *condor_schedd* daemon describes its state. ClassAd attributes that appear in a Scheduler ClassAd are listed and described in the unnumbered subsection labeled Scheduler ClassAd Attributes on page 921.

`Negotiator` Each *condor_negotiator* daemon describes its state. ClassAd attributes that appear

in a Negotiator ClassAd are listed and described in the unnumbered subsection labeled Negotiator ClassAd Attributes on page 923.

Collector Each *condor_collector* daemon describes its state. ClassAd attributes that appear in a Collector ClassAd are listed and described in the unnumbered subsection labeled Collector ClassAd Attributes on page 924.

Query This section has not yet been written

Job ClassAd Attributes

AllRemoteHosts: String containing a comma-separated list of all the remote machines running a parallel or mpi universe job.

Args: String representing the arguments passed to the job.

CkptArch: String describing the architecture of the machine this job executed on at the time it last produced a checkpoint. If the job has never produced a checkpoint, this attribute is `undefined`.

CkptOpSys: String describing the operating system of the machine this job executed on at the time it last produced a checkpoint. If the job has never produced a checkpoint, this attribute is `undefined`.

ClusterId: Integer cluster identifier for this job. A cluster is a group of jobs that were submitted together. Each job has its own unique job identifier within the cluster, but shares a common cluster identifier. The value changes each time a job or set of jobs are queued for execution under Condor.

Cmd: The path to and the file name of the job to be executed.

ConcurrencyLimits: A string list, delimited by commas and space characters. The items in the list identify named resources that the job requires.

CommittedTime: The number of seconds of wall clock time that the job has been allocated a machine, excluding the time spent on run attempts that were evicted without a checkpoint. Like `RemoteWallClockTime`, this includes time the job spent in a suspended state, so the total committed wall time spent running is

```
CommittedTime - CommittedSuspensionTime
```

CommittedSlotTime: This attribute is identical to `CommittedTime` except that the time is multiplied by the `SlotWeight` of the machine(s) that ran the job. This relies on `SlotWeight` being listed in `SYSTEM_JOB_MACHINE_ATTRS`.

CumulativeSlotTime: This attribute is identical to `RemoteWallClockTime` except that the time is multiplied by the `SlotWeight` of the machine(s) that ran the job. This relies on `SlotWeight` being listed in `SYSTEM_JOB_MACHINE_ATTRS`.

CompletionDate: The time when the job completed, or the value 0 if the job has not yet completed. Measured in the number of seconds since the epoch (00:00:00 UTC, Jan 1, 1970).

CommittedSuspensionTime: A running total of the number of seconds the job has spent in suspension during time in which the job was not evicted without a checkpoint. This number is updated when the job is checkpointed and when it exits.

CumulativeSuspensionTime: A running total of the number of seconds the job has spent in suspension for the life of the job.

CurrentHosts: The number of hosts in the claimed state, due to this job.

DAGManJobId: For a DAGMan node job only, the `ClusterId` job ClassAd attribute of the *condor_dagman* job which is the parent of this node job. It is only one layer deep for nested DAGs.

DAGParentNodeNames: For a DAGMan node job only, a comma separated list of each *JobName* which is a parent node of this job's node. This attribute is passed through to the job via the *condor_submit* command line, if it does not exceed the line length defined with `_POSIX_ARG_MAX`. For example, if a node job has two parents with *JobName*s B and C, the *condor_submit* command line will contain

```
-append +DAGParentNodeNames=B,C
```

DeltacloudAvailableActions: Used for grid-type deltacloud jobs. For a running job, Condor sets this string to contain a comma-separated list of actions that can be performed on a Deltacloud instance, as given by the selected service.

DeltacloudHardwareProfile: String taken from the submit description file command **deltacloud_hardware_profile**. Specifies the hardware configuration to be used for a grid-type deltacloud job.

DeltacloudHardwareProfileCpu: String taken from the submit description file command **deltacloud_hardware_profile_cpu**. Specifies CPU details in the hardware configuration to be used for a grid-type deltacloud job.

DeltacloudHardwareProfileMemory: String taken from the submit description file command **deltacloud_hardware_profile_memory**. Specifies memory (RAM) details in the hardware configuration to be used for a grid-type deltacloud job.

DeltacloudHardwareProfileStorage: String taken from the submit description file command **deltacloud_hardware_profile_storage**. Specifies memory (disk) details in the hardware configuration to be used for a grid-type deltacloud job.

DeltacloudImageId: String taken from the submit description file command **deltacloud_image_id**. Specifies the virtual machine image to use for a grid-type deltacloud job.

DeltacloudKeyname: String taken from the submit description file command **deltacloud_keyname**. Specifies the SSH key pair to use for a grid-type deltacloud job.

DeltacloudPasswordFile: String taken from the submit description file command **deltacloud_password_file**. Specifies a file containing the secret key to be used to authenticate with the Deltacloud service for a grid-type deltacloud job.

DeltacloudPrivateNetworkAddresses: For a running Deltacloud instance, Condor receives and sets this comma-separated list of the private IP addresses allocated to the running virtual machine.

DeltacloudPublicNetworkAddresses: For a running Deltacloud instance, Condor receives and sets this comma-separated list of the public IP addresses allocated to the running virtual machine.

DeltacloudRealmId: String taken from the submit description file command **deltacloud_ream_id**. Specifies the realm to be used for a grid-type deltacloud job.

DeltacloudUserData: String taken from the submit description file command **deltacloud_user_data**. Specifies a block of data to be provided to the instance for a grid-type deltacloud job.

DeltacloudUsername: String taken from the submit description file command **deltacloud_username**. Specifies the user name to be used to authenticate with the Deltacloud service for a grid-type deltacloud job.

DiskUsage: Amount of disk space (Kbytes) in the Condor execute directory on the execute machine that this job has used. The initial estimate may be specified in the job submit file using the **request_disk** command. If not initialized by the job's request, **vm** universe jobs will default to an initial value of the disk image size. If not initialized by the job's request, non-**vm** universe jobs will default to an initial value of the sum of the job's executable and all input files.

EmailAttributes: A string containing a comma-separated list of job ClassAd attributes. For each attribute name in the list, its value will be included in the e-mail notification upon job completion.

EnteredCurrentStatus: An integer containing the epoch time of when the job entered into its current status So for example, if the job is on hold, the ClassAd expression

```
CurrentTime - EnteredCurrentStatus
```

will equal the number of seconds that the job has been on hold.

ExecutableSize: Size of the executable in Kbytes.

ExitBySignal: An attribute that is `True` when a user job exits via a signal and `False` otherwise. For some grid universe jobs, how the job exited is unavailable. In this case, `ExitBySignal` is set to `False`.

ExitCode: When a user job exits by means other than a signal, this is the exit return code of the user job. For some grid universe jobs, how the job exited is unavailable. In this case, `ExitCode` is set to 0.

ExitSignal: When a user job exits by means of an unhandled signal, this attribute takes on the numeric value of the signal. For some grid universe jobs, how the job exited is unavailable. In this case, `ExitSignal` will be undefined.

ExitStatus: The way that Condor previously dealt with a job's exit status. This attribute should no longer be used. It is not always accurate in heterogeneous pools, or if the job exited with a signal. Instead, see the attributes: `ExitBySignal`, `ExitCode`, and `ExitSignal`.

GridJobStatus: A string containing the job's status as reported by the remote job management system.

GridResource: A string defined by the right hand side of the the submit description file command **grid_resource**. It specifies the target grid type, plus additional parameters specific to the grid type.

HoldKillSig: Currently only for scheduler and local universe jobs, a string containing a name of a signal to be sent to the job if the job is put on hold.

HoldReason: A string containing a human-readable message about why a job is on hold. This is the message that will be displayed in response to the command `condor_q -hold`. It can be used to determine if a job should be released or not.

HoldReasonCode: An integer value that represents the reason that a job was put on hold.

HoldReasonSubCode: An integer value that represents further information to go along with the `HoldReasonCode`, for some values of `HoldReasonCode`. See `HoldReasonCode` for the values.

HookKeyword: A string that uniquely identifies a set of job hooks, and added to the ClassAd once a job is fetched.

ImageSize: Estimate of the memory image size of the job in Kbytes. The initial estimate may be specified in the job submit file using **request_memory** or for **vm** universe jobs using **vm_memory**. Otherwise, the initial value is equal to the size of the executable for non-vm universe jobs, and 0 for vm universe jobs. When the job writes a checkpoint, the `ImageSize` attribute is set to the size of the checkpoint file (since the checkpoint file contains the job's memory image). A vanilla universe job's `ImageSize` is recomputed internally every 15 seconds.

IwdFlushNFSCache: A boolean expression that controls whether or not Condor attempts to flush a submit machine's NFS cache, in order to refresh a Condor job's initial working directory. The value will be `True`, unless a job explicitly adds this attribute, setting it to `False`.

JobAdInformationAttrs: A comma-separated list of attribute names. The named attributes and their values are written in the user log whenever any event is being written to the log. This is the same as the configuration setting `EVENT_LOG_INFORMATION_ATTRS` (see page 175) but it applies to the user log instead of the system event log.

Integer Code	*Reason for Hold*	*HoldReasonSubCode*
1	The user put the job on hold with *condor_hold*.	
2	Globus middleware reported an error.	The GRAM error number.
3	The `PERIODIC_HOLD` expression evaluated to `True`.	
4	The credentials for the job are invalid.	
5	A job policy expression evaluated to `Undefined`.	
6	The *condor_starter* failed to start the executable.	The Unix error number.
7	The standard output file for the job could not be opened.	The Unix error number.
8	The standard input file for the job could not be opened.	The Unix error number.
9	The standard output stream for the job could not be opened.	The Unix error number.
10	The standard input stream for the job could not be opened.	The Unix error number.
11	An internal Condor protocol error was encountered when transferring files.	
12	The *condor_starter* failed to download input files.	The Unix error number.
13	The *condor_starter* failed to upload output files.	The Unix error number.
14	The initial working directory of the job cannot be accessed.	The Unix error number.
15	The user requested the job be submitted on hold.	
16	Input files are being spooled.	
17	A standard universe job is not compatible with the *condor_shadow* version available on the submitting machine.	
18	An internal Condor protocol error was encountered when transferring files.	
19	`<Keyword>_HOOK_PREPARE_JOB` was defined but could not be executed or returned failure.	
20	The job missed its deferred execution time and therefore failed to run.	
21	The job was put on hold because `WANT_HOLD` in the machine policy was true.	
22	Unable to initialize user log.	

`JobLeaseDuration`: The number of seconds set for a job lease, the amount of time that a job may continue running on a remote resource, despite its submitting machine's lack of response. See section 2.14.4 for details on job leases.

`JobPrio`: Integer priority for this job, set by *condor_submit* or *condor_prio*. The default value is 0. The higher the number, the greater (better) the priority.

`JobRunCount`: This attribute is retained for backwards compatibility. It may go away in the future. It is equivalent to `NumShadowStarts` for all universes except **scheduler**. For the **scheduler** universe, this attribute is equivalent to `NumJobStarts`.

`JobStartDate`: Time at which the job first began running. Measured in the number of seconds since the epoch (00:00:00 UTC, Jan 1, 1970).

`JobStatus`: Integer which indicates the current status of the job.

Value	*Status*
0	Unexpanded (the job has never run)
1	Idle
2	Running
3	Removed
4	Completed
5	Held
6	Transferring Output

JobUniverse: Integer which indicates the job universe.

Value	*Universe*
1	standard
5	vanilla
7	scheduler
8	MPI
9	grid
10	java
11	parallel
12	local
13	vm

KillSig: The Unix signal number that the job wishes to be sent before being forcibly killed. It is relevant only for jobs running on Unix machines.

KillSigTimeout: The number of seconds that the job (other than the standard universe) requests the *condor_starter* wait after sending the signal defined as `KillSig` and before forcibly removing the job. The actual amount of time will be the minimum of this value and the execute machine's configuration variable `KILLING_TIMEOUT` .

LastCheckpointPlatform: An opaque string which is the `CheckpointPlatform` identifier from the last machine where this standard universe job had successfully produced a checkpoint.

LastCkptServer: Host name of the last checkpoint server used by this job. When a pool is using multiple checkpoint servers, this tells the job where to find its checkpoint file.

LastCkptTime: Time at which the job last performed a successful checkpoint. Measured in the number of seconds since the epoch (00:00:00 UTC, Jan 1, 1970).

LastMatchTime: An integer containing the epoch time when the job was last successfully matched with a resource (gatekeeper) Ad.

LastRejMatchReason: If, at any point in the past, this job failed to match with a resource ad, this attribute will contain a string with a human-readable message about why the match failed.

LastRejMatchTime: An integer containing the epoch time when Condor-G last tried to find a match for the job, but failed to do so.

LastSuspensionTime: Time at which the job last performed a successful suspension. Measured in the number of seconds since the epoch (00:00:00 UTC, Jan 1, 1970).

LastVacateTime: Time at which the job was last evicted from a remote workstation. Measured in the number of seconds since the epoch (00:00:00 UTC, Jan 1, 1970).

LeaveJobInQueue: A boolean expression that defaults to `False`, causing the job to be removed from the queue upon completion. An exception is if the job is submitted using `condor_submit -spool`. For this case, the default expression causes the job to be kept in the queue for 10 days after completion.

LocalSysCpu: An accumulated number of seconds of system CPU time that the job caused to the machine upon which the job was submitted.

LocalUserCpu: An accumulated number of seconds of user CPU time that the job caused to the machine upon which the job was submitted.

MachineAttr<X><N>: Machine attribute of name `<X>` that is placed into this job ClassAd, as specified by the configuration variable `SYSTEM_JOB_MACHINE_ATTRS`. With the potential for multiple run attempts, `<N>` represents an integer value providing historical values of this machine attribute for multiple runs. The most recent run will have a value of `<N>` equal to `0`. The next most recent run will have a value of `<N>` equal to `1`.

MaxHosts: The maximum number of hosts that this job would like to claim. As long as `CurrentHosts` is the same as `MaxHosts`, no more hosts are negotiated for.

MaxJobRetirementTime: Maximum time in seconds to let this job run uninterrupted before kicking it off when it is being preempted. This can only decrease the amount of time from what the corresponding startd expression allows.

MinHosts: The minimum number of hosts that must be in the claimed state for this job, before the job may enter the running state.

NextJobStartDelay: An integer number of seconds delay time after this job starts until the next job is started. The value is limited by the configuration variable `MAX_NEXT_JOB_START_DELAY`.

NiceUser: Boolean value which when `True` indicates that this job is a *nice* job, raising its user priority value, thus causing it to run on a machine only when no other Condor jobs want the machine.

NTDomain: A string that identifies the NT domain under which a job's owner authenticates on a platform running Windows.

NumCkpts: A count of the number of checkpoints written by this job during its lifetime.

NumGlobusSubmits: An integer that is incremented each time the *condor_gridmanager* receives confirmation of a successful job submission into Globus.

NumJobMatches: An integer that is incremented by the *condor_schedd* each time the job is matched with a resource ad by the negotiator.

NumJobStarts: An integer count of the number of times the job started executing. This is not (yet) defined for **standard** universe jobs.

NumJobReconnects: An integer count of the number of times a job successfully reconnected after being disconnected. This occurs when the *condor_shadow* and *condor_starter* lose contact, for example because of transient network failures or a *condor_shadow* or *condor_schedd* restart. This attribute is only defined for jobs that can reconnected: those in the **vanilla** and **java** universes.

NumPids: A count of the number of child processes that this job has.

NumRestarts: A count of the number of restarts from a checkpoint attempted by this job during its lifetime.

NumShadowExceptions: An integer count of the number of times the *condor_shadow* daemon had a fatal error for a given job.

NumShadowStarts: An integer count of the number of times a *condor_shadow* daemon was started for a given job. This attribute is not defined for **scheduler** universe jobs, since they do not have a *condor_shadow* daemon associated with them. For **local** universe jobs, this attribute *is* defined, even though the process that manages the job is technically a *condor_starter* rather than a *condor_shadow*. This keeps the management of the local universe and other universes as similar as possible.

NumSystemHolds: An integer that is incremented each time Condor-G places a job on hold due to some sort of error condition. This counter is useful, since Condor-G will always place a job on hold when it gives up on some error condition. Note that if the user places the job on hold using the *condor_hold* command, this attribute is not incremented.

OtherJobRemoveRequirements: A string that defines a list of jobs. When the job with this attribute defined is removed, all other jobs defined by the list are also removed. The string is an expression that defines a constraint equivalent to the one implied by the command

```
condor_rm -constraint <constraint>
```

This attribute is used for jobs managed with *condor_dagman* to ensure that node jobs of the DAG are removed when the *condor_dagman* job itself is removed. Note that the list of jobs defined by this attribute must not form a cyclic removal of jobs, or the *condor_schedd* will go into an infinite loop when any of the jobs is removed.

Owner: String describing the user who submitted this job.

ParallelShutdownPolicy: A string that is only relevant to parallel universe jobs. Without this attribute defined, the default policy applied to parallel universe jobs is to consider the whole job completed when the first node exits, killing processes running on all remaining nodes. If defined to the following strings, Condor's behavior changes:

"WAIT_FOR_ALL" Condor will wait until every node in the parallel job has completed to consider the job finished.

ProcId: Integer process identifier for this job. Within a cluster of many jobs, each job has the same `ClusterId`, but will have a unique `ProcId`. Within a cluster, assignment of a `ProcId` value will start with the value 0. The job (process) identifier described here is unrelated to operating system PIDs.

QDate: Time at which the job was submitted to the job queue. Measured in the number of seconds since the epoch (00:00:00 UTC, Jan 1, 1970).

ReleaseReason: A string containing a human-readable message about why the job was released from hold.

RemoteIwd: The path to the directory in which a job is to be executed on a remote machine.

RemoteSysCpu: The total number of seconds of system CPU time (the time spent at system calls) the job used on remote machines. This does not count time spent on run attempts that were evicted without a checkpoint.

RemoteUserCpu: The total number of seconds of user CPU time the job used on remote machines. This does not count time spent on run attempts that were evicted without a checkpoint.

RemoteWallClockTime: Cumulative number of seconds the job has been allocated a machine. This also includes time spent in suspension (if any), so the total real time spent running is

```
RemoteWallClockTime - CumulativeSuspensionTime
```

Note that this number does not get reset to zero when a job is forced to migrate from one machine to another. `CommittedTime`, on the other hand, is just like `RemoteWallClockTime` except it does get reset to 0 whenever the job is evicted without a checkpoint.

RemoveKillSig: Currently only for scheduler universe jobs, a string containing a name of a signal to be sent to the job if the job is removed.

ResidentSetSize: Estimate of the physical memory in use by the job in Kbytes while it is running.

StageOutFinish: An attribute representing a Unix epoch time that is defined for a job that is spooled to a remote site using `condor_submit -spool` or Condor-C and causes Condor to hold the output in the spool while the job waits in the queue in the `Completed` state. This attribute is defined when retrieval of the output finishes.

StageOutStart: An attribute representing a Unix epoch time that is defined for a job that is spooled to a remote site using `condor_submit -spool` or Condor-C and causes Condor to hold the output in the spool while the job waits in the queue in the `Completed` state. This attribute is defined when retrieval of the output begins.

StreamErr: An attribute utilized only for grid universe jobs. The default value is `True`. If `True`, and `TransferErr` is `True`, then standard error is streamed back to the submit machine, instead of doing the transfer (as a whole) after the job completes. If `False`, then standard error is transferred back to the submit machine (as a whole) after the job completes. If `TransferErr` is `False`, then this job attribute is ignored.

StreamOut: An attribute utilized only for grid universe jobs. The default value is `True`. If `True`, and `TransferOut` is `True`, then job output is streamed back to the submit machine, instead of doing the transfer (as a whole) after the job completes. If `False`, then job output is transferred back to the submit machine (as a whole) after the job completes. If `TransferOut` is `False`, then this job attribute is ignored.

TotalSuspensions: A count of the number of times this job has been suspended during its lifetime.

TransferErr: An attribute utilized only for grid universe jobs. The default value is `True`. If `True`, then the error output from the job is transferred from the remote machine back to the submit machine. The name of the file after transfer is the file referred to by job attribute `Err`. If `False`, no transfer takes place (remote to submit machine), and the name of the file is the file referred to by job attribute `Err`.

TransferExecutable: An attribute utilized only for grid universe jobs. The default value is `True`. If `True`, then the job executable is transferred from the submit machine to the remote machine. The name of the file (on the submit machine) that is transferred is given by the job attribute `Cmd`. If `False`, no transfer takes place, and the name of the file used (on the remote machine) will be as given in the job attribute `Cmd`.

TransferIn: An attribute utilized only for grid universe jobs. The default value is `True`. If `True`, then the job input is transferred from the submit machine to the remote machine. The name of the file that is transferred is given by the job attribute `In`. If `False`, then the job's input is taken from a file on the remote machine (pre-staged), and the name of the file is given by the job attribute `In`.

TransferOut: An attribute utilized only for grid universe jobs. The default value is `True`. If `True`, then the output from the job is transferred from the remote machine back to the submit machine. The name of the file after transfer is the file referred to by job attribute `Out`. If `False`, no transfer takes place (remote to submit machine), and the name of the file is the file referred to by job attribute `Out`.

WindowsBuildNumber: An integer, extracted from the platform type of the machine upon which this job is submitted, representing a build number for a Windows operating system. This attribute only exists for jobs submitted from Windows machines.

WindowsMajorVersion: An integer, extracted from the platform type of the machine upon which this job is submitted, representing a major version number (currently 5 or 6) for a Windows operating system. This attribute only exists for jobs submitted from Windows machines.

WindowsMinorVersion: An integer, extracted from the platform type of the machine upon which this job is submitted, representing a minor version number (currently 0, 1, or 2) for a Windows operating system. This attribute only exists for jobs submitted from Windows machines.

X509UserProxyExpiration: For a job that defines the submit description file command **x509userproxy**, this is the time at which the indicated X.509 proxy credential will expire, measured in the number of seconds since the epoch (00:00:00 UTC, Jan 1, 1970).

X509UserProxyFirstFQAN: For a vanilla or grid universe job that defines the submit description file command **x509userproxy**, this is the VOMS Fully Qualified Attribute Name (FQAN) of the primary role of the credential. A credential may have multiple roles defined, but by convention the one listed first is the primary role.

X509UserProxyFQAN: For a vanilla or grid universe job that defines the submit description file command **x509userproxy**, this is a serialized list of the DN and all FQAN. A comma is used as a separator, and any existing commas in the DN or FQAN are replaced with the string `&comma;`. Likewise, any ampersands in the DN or FQAN are replaced with `&`.

X509UserProxySubject: For a vanilla or grid universe job that defines the submit description file command **x509userproxy**, this attribute contains the Distinguished Name (DN) of the credential used to submit the job.

X509UserProxyVOName: For a vanilla or grid universe job that defines the submit description file command **x509userproxy**, this is the name of the VOMS virtual organization (VO) that the user's credential is part of.

DelegateJobGSICredentialsLifetime: An integer that specifies the maximum number of seconds for which delegated proxies should be valid. The default behavior is determined by the configuration setting `DELEGATE_JOB_GSI_CREDENTIALS_LIFETIME` , which defaults to one day. A value of 0 indicates that the delegated proxy should be valid for as long as allowed by the credential used to create the proxy. This setting currently only applies to proxies delegated for non-grid jobs and Condor-C jobs. It does not currently apply to globus grid jobs, which always behave as though this setting were 0. This setting has no effect if the configuration setting `DELEGATE_JOB_GSI_CREDENTIALS` is false, because in that case the job proxy is copied rather than delegated.

The following job ClassAd attributes are relevant only for **vm** universe jobs.

VM_MACAddr: The MAC address of the virtual machine's network interface, in the standard format of six groups of two hexadecimal digits separated by colons. This attribute is currently limited to apply only to Xen virtual machines.

Machine ClassAd Attributes

Activity: String which describes Condor job activity on the machine. Can have one of the following values:

"Idle": There is no job activity

"Busy": A job is busy running

"Suspended": A job is currently suspended

"Vacating": A job is currently checkpointing

"Killing": A job is currently being killed

"Benchmarking": The startd is running benchmarks

Arch: String with the architecture of the machine. Currently supported architectures have the following string definitions:

"INTEL": Intel x86 CPU (Pentium, Xeon, etc).

"X86_64": AMD/Intel 64-bit X86

These strings show definitions for architectures no longer suppported:

"IA64": Intel Itanium

"SUN4u": Sun UltraSparc CPU

"SUN4x": A Sun Sparc CPU other than an UltraSparc, i.e. sun4m or sun4c CPU found in older Sparc workstations such as the Sparc 10, Sparc 20, IPC, IPX, etc.

"PPC": 32-bit PowerPC

"PPC64": 64-bit PowerPC

CheckpointPlatform: A string which opaquely encodes various aspects about a machine's operating system, hardware, and kernel attributes. It is used to identify systems where previously taken checkpoints for the standard universe may resume.

ClockDay: The day of the week, where 0 = Sunday, 1 = Monday, . . ., 6 = Saturday.

ClockMin: The number of minutes passed since midnight.

CondorLoadAvg: The portion of the load average generated by Condor (either from remote jobs or running benchmarks).

ConsoleIdle: The number of seconds since activity on the system console keyboard or console mouse has last been detected.

Cpus: Number of CPUs in this machine, i.e. 1 = single CPU machine, 2 = dual CPUs, etc.

CurrentRank: A float which represents this machine owner's affinity for running the Condor job which it is currently hosting. If not currently hosting a Condor job, `CurrentRank` is 0.0. When a machine is claimed, the attribute's value is computed by evaluating the machine's `Rank` expression with respect to the current job's ClassAd.

Disk: The amount of disk space on this machine available for the job in Kbytes (e.g. 23000 = 23 megabytes). Specifically, this is the amount of disk space available in the directory specified in the Condor configuration files by the `EXECUTE` macro, minus any space reserved with the `RESERVED_DISK` macro.

DynamicSlot: For SMP machines that allow dynamic partitioning of a slot, this boolean value identifies that this dynamic slot may be partitioned.

EnteredCurrentActivity: Time at which the machine entered the current Activity (see `Activity` entry above). On all platforms (including NT), this is measured in the number of integer seconds since the Unix epoch (00:00:00 UTC, Jan 1, 1970).

FileSystemDomain: A domain name configured by the Condor administrator which describes a cluster of machines which all access the same, uniformly-mounted, networked file systems usually via NFS or AFS. This is useful for Vanilla universe jobs which require remote file access.

HasVM: A boolean value added to the machine ClassAd when the configuration triggers the detection of virtual machine software.

JobVM_VCPUS: An attribute defined if a vm universe job is running on this slot. Defined by the number of virtualized CPUs in the virtual machine.

KeyboardIdle: The number of seconds since activity on any keyboard or mouse associated with this machine has last been detected. Unlike `ConsoleIdle`, `KeyboardIdle` also takes activity on pseudo-terminals into account (i.e. virtual "keyboard" activity from telnet and rlogin sessions as well). Note that `KeyboardIdle` will always be equal to or less than `ConsoleIdle`.

KFlops: Relative floating point performance as determined via a Linpack benchmark.

LastHeardFrom: Time when the Condor central manager last received a status update from this machine. Expressed as the number of integer seconds since the Unix epoch (00:00:00 UTC, Jan 1, 1970). Note: This attribute is only inserted by the central manager once it receives the ClassAd. It is not present in the *condor_startd* copy of the ClassAd. Therefore, you could not use this attribute in defining *condor_startd* expressions (and you would not want to).

LoadAvg: A floating point number with the machine's current load average.

Machine: A string with the machine's fully qualified host name.

Memory: The amount of RAM in megabytes.

Mips: Relative integer performance as determined via a Dhrystone benchmark.

MonitorSelfAge: The number of seconds that this daemon has been running.

MonitorSelfCPUUsage: The fraction of recent CPU time utilized by this daemon.

MonitorSelfImageSize: The amount of virtual memory consumed by this daemon in Kbytes.

MonitorSelfRegisteredSocketCount: The current number of sockets registered by this daemon.

MonitorSelfResidentSetSize: The amount of resident memory used by this daemon in Kbytes.

MonitorSelfSecuritySessions: The number of open (cached) security sessions for this daemon.

MonitorSelfTime: The time, represented as the number of second elapsed since the Unix epoch (00:00:00 UTC, Jan 1, 1970), at which this daemon last checked and set the attributes with names that begin with the string `MonitorSelf`.

MyAddress: String with the IP and port address of the *condor_startd* daemon which is publishing this machine ClassAd. When using CCB, *condor_shared_port*, and/or an additional private network interface, that information will be included here as well.

MyType: The ClassAd type; always set to the literal string `"Machine"`.

Name: The name of this resource; typically the same value as the `Machine` attribute, but could be customized by the site administrator. On SMP machines, the *condor_startd* will divide the CPUs up into separate slots, each with with a unique name. These names will be of the form "slot#@full.hostname", for example, "slot1@vulture.cs.wisc.edu", which signifies slot number 1 from vulture.cs.wisc.edu.

OpSys: String describing the operating system running on this machine. Currently supported operating systems have the following string definitions:

"LINUX": for LINUX 2.0.x, LINUX 2.2.x, LINUX 2.4.x, or LINUX 2.6.x kernel systems

"OSX": for Darwin

"WINNT50": for Windows 2000

"WINNT51": for Windows XP

"WINNT52": for Windows Server 2003

"WINNT60": for Windows Vista

"WINNT61": for Windows 7

These strings show definitions for operating systems no longer suppported:

"SOLARIS28": for Solaris 2.8 or 5.8

"SOLARIS29": for Solaris 2.9 or 5.9

Requirements: A boolean, which when evaluated within the context of the machine ClassAd and a job ClassAd, must evaluate to TRUE before Condor will allow the job to use this machine.

MaxJobRetirementTime: An expression giving the maximum time in seconds that the startd will wait for the job to finish before kicking it off if it needs to do so. This is evaluated in the context of the job ClassAd, so it may refer to job attributes as well as machine attributes.

PartitionableSlot: For SMP machines, a boolean value identifying that this slot may be partitioned.

SlotID: For SMP machines, the integer that identifies the slot. The value will be `X` for the slot with

```
name="slotX@full.hostname"
```

For non-SMP machines with one slot, the value will be 1. NOTE: This attribute was added in Condor version 6.9.3. For older versions of Condor, see `VirtualMachineID` below.

SlotWeight: This specifies the weight of the slot when calculating usage, computing fair shares, and enforcing group quotas. For example, claiming a slot with `SlotWeight = 2` is equivalent to claiming two `SlotWeight = 1` slots. See the description of `SlotWeight` on page 200.

StartdIpAddr: String with the IP and port address of the *condor_startd* daemon which is publishing this machine ClassAd. When using CCB, *condor_shared_port*, and/or an additional private network interface, that information will be included here as well.

State: String which publishes the machine's Condor state. Can be:

"Owner": The machine owner is using the machine, and it is unavailable to Condor.

"Unclaimed": The machine is available to run Condor jobs, but a good match is either not available or not yet found.

"Matched": The Condor central manager has found a good match for this resource, but a Condor scheduler has not yet claimed it.

"Claimed": The machine is claimed by a remote *condor_schedd* and is probably running a job.

"Preempting": A Condor job is being preempted (possibly via checkpointing) in order to clear the machine for either a higher priority job or because the machine owner wants the machine back.

TargetType: Describes what type of ClassAd to match with. Always set to the string literal `"Job"`, because machine ClassAds always want to be matched with jobs, and vice-versa.

TotalTimeBackfillBusy: The number of seconds that this machine (slot) has accumulated within the backfill busy state and activity pair since the *condor_startd* began executing. This attribute will only be defined if it has a value greater than 0.

TotalTimeBackfillIdle: The number of seconds that this machine (slot) has accumulated within the backfill idle state and activity pair since the *condor_startd* began executing. This attribute will only be defined if it has a value greater than 0.

TotalTimeBackfillKilling: The number of seconds that this machine (slot) has accumulated within the backfill killing state and activity pair since the *condor_startd* began executing. This attribute will only be defined if it has a value greater than 0.

TotalTimeClaimedBusy: The number of seconds that this machine (slot) has accumulated within the claimed busy state and activity pair since the *condor_startd* began executing. This attribute will only be defined if it has a value greater than 0.

TotalTimeClaimedIdle: The number of seconds that this machine (slot) has accumulated within the claimed idle state and activity pair since the *condor_startd* began executing. This attribute will only be defined if it has a value greater than 0.

TotalTimeClaimedRetiring: The number of seconds that this machine (slot) has accumulated within the claimed retiring state and activity pair since the *condor_startd* began executing. This attribute will only be defined if it has a value greater than 0.

TotalTimeClaimedSuspended: The number of seconds that this machine (slot) has accumulated within the claimed suspended state and activity pair since the *condor_startd* began executing. This attribute will only be defined if it has a value greater than 0.

TotalTimeMatchedIdle: The number of seconds that this machine (slot) has accumulated within the matched idle state and activity pair since the *condor_startd* began executing. This attribute will only be defined if it has a value greater than 0.

TotalTimeOwnerIdle: The number of seconds that this machine (slot) has accumulated within the owner idle state and activity pair since the *condor_startd* began executing. This attribute will only be defined if it has a value greater than 0.

TotalTimePreemptingKilling: The number of seconds that this machine (slot) has accumulated within the preempting killing state and activity pair since the *condor_startd* began executing. This attribute will only be defined if it has a value greater than 0.

TotalTimePreemptingVacating: The number of seconds that this machine (slot) has accumulated within the preempting vacating state and activity pair since the *condor_startd* began executing. This attribute will only be defined if it has a value greater than 0.

TotalTimeUnclaimedBenchmarking: The number of seconds that this machine (slot) has accumulated within the unclaimed benchmarking state and activity pair since the *condor_startd* began executing. This attribute will only be defined if it has a value greater than 0.

TotalTimeUnclaimedIdle: The number of seconds that this machine (slot) has accumulated within the unclaimed idle state and activity pair since the *condor_startd* began executing. This attribute will only be defined if it has a value greater than 0.

UidDomain: a domain name configured by the Condor administrator which describes a cluster of machines which all have the same `passwd` file entries, and therefore all have the same logins.

VirtualMachineID: Starting with Condor version 6.9.3, this attribute is now longer used. Instead, use `SlotID`, as described above. This will only be present if `ALLOW_VM_CRUFT` is TRUE.

VirtualMemory: The amount of currently available virtual memory (swap space) expressed in Kbytes.

VM_AvailNum: The maximum number of vm universe jobs that can be started on this machine. This maximum is set by the configuration variable VM_MAX_NUMBER .

VM_Guest_Mem: An attribute defined if a vm universe job is running on this slot. Defined by the amount of memory in use by the virtual machine, given in Mbytes.

VM_Memory: Gives the amount of memory available for starting additional VM jobs on this machine, given in Mbytes. The maximum value is set by the configuration variable VM_MEMORY .

VM_Networking: A boolean value indicating whether networking is allowed for virtual machines on this machine.

VM_Type: The type of virtual machine software that can run on this machine. The value is set by the configuration variable VM_TYPE .

WindowsBuildNumber: An integer, extracted from the platform type, representing a build number for a Windows operating system. This attribute only exists on Windows machines.

WindowsMajorVersion: An integer, extracted from the platform type, representing a major version number (currently 5 or 6) for a Windows operating system. This attribute only exists on Windows machines.

WindowsMinorVersion: An integer, extracted from the platform type, representing a minor version number (currently 0, 1, or 2) for a Windows operating system. This attribute only exists on Windows machines.

In addition, there are a few attributes that are automatically inserted into the machine ClassAd whenever a resource is in the Claimed state:

ClientMachine: The host name of the machine that has claimed this resource

RemoteOwner: The name of the user who originally claimed this resource.

RemoteUser: The name of the user who is currently using this resource. In general, this will always be the same as the RemoteOwner, but in some cases, a resource can be claimed by one entity that hands off the resource to another entity which uses it. In that case, RemoteUser would hold the name of the entity currently using the resource, while RemoteOwner would hold the name of the entity that claimed the resource.

PreemptingOwner: The name of the user who is preempting the job that is currently running on this resource.

PreemptingUser: The name of the user who is preempting the job that is currently running on this resource. The relationship between PreemptingUser and PreemptingOwner is the same as the relationship between RemoteUser and RemoteOwner.

PreemptingRank: A float which represents this machine owner's affinity for running the Condor job which is waiting for the current job to finish or be preempted. If not currently hosting a Condor job, `PreemptingRank` is undefined. When a machine is claimed and there is already a job running, the attribute's value is computed by evaluating the machine's `Rank` expression with respect to the preempting job's ClassAd.

TotalClaimRunTime: A running total of the amount of time (in seconds) that all jobs (under the same claim) ran (have spent in the Claimed/Busy state).

TotalClaimSuspendTime: A running total of the amount of time (in seconds) that all jobs (under the same claim) have been suspended (in the Claimed/Suspended state).

TotalJobRunTime: A running total of the amount of time (in seconds) that a single job ran (has spent in the Claimed/Busy state).

TotalJobSuspendTime: A running total of the amount of time (in seconds) that a single job has been suspended (in the Claimed/Suspended state).

There are a few attributes that are only inserted into the machine ClassAd if a job is currently executing. If the resource is claimed but no job are running, none of these attributes will be defined.

JobId: The job's identifier (for example, `152.3`), as seen from *condor_q* on the submitting machine.

JobStart: The time stamp in integer seconds of when the job began executing, since the Unix epoch (00:00:00 UTC, Jan 1, 1970). For idle machines, the value is `UNDEFINED`.

LastPeriodicCheckpoint: If the job has performed a periodic checkpoint, this attribute will be defined and will hold the time stamp of when the last periodic checkpoint was begun. If the job has yet to perform a periodic checkpoint, or cannot checkpoint at all, the `LastPeriodicCheckpoint` attribute will not be defined.

There are a few attributes that are applicable to machines that are offline, that is, hibernating.

MachineLastMatchTime: The Unix epoch time when this offline ad would have been matched to a job if the machine were online. In addition, the slot1 ad of a multi-slot machine will have `slot<X>_MachineLastMatchTime` where `X` is replaced by the slot id of each of the slots with `MachineLastMatchTime` defined.

Offline: true for ads that are in an offline state in the collector. Such ads are stored persistently so that they will still be there after the collector restarts.

Unhibernate: a boolean expression that specifies when a hibernating machine should be woken up (e.g. by *condor_rooster*).

Finally, the single attribute, `CurrentTime`, is defined by the ClassAd environment.

CurrentTime: Evaluates to the the number of integer seconds since the Unix epoch (00:00:00 UTC, Jan 1, 1970).

DaemonMaster ClassAd Attributes

CkptServer: A string with with the fully qualified host name of the machine running a checkpoint server.

DaemonStartTime: The time that this daemon was started, represented as the number of second elapsed since the Unix epoch (00:00:00 UTC, Jan 1, 1970).

Machine: A string with the machine's fully qualified host name.

MasterIpAddr: String with the IP and port address of the *condor_master* daemon which is publishing this DaemonMaster ClassAd.

MonitorSelfAge: The number of seconds that this daemon has been running.

MonitorSelfCPUUsage: The fraction of recent CPU time utilized by this daemon.

MonitorSelfImageSize: The amount of virtual memory consumed by this daemon in Kbytes.

MonitorSelfRegisteredSocketCount: The current number of sockets registered by this daemon.

MonitorSelfResidentSetSize: The amount of resident memory used by this daemon in Kbytes.

MonitorSelfSecuritySessions: The number of open (cached) security sessions for this daemon.

MonitorSelfTime: The time, represented as the number of second elapsed since the Unix epoch (00:00:00 UTC, Jan 1, 1970), at which this daemon last checked and set the attributes with names that begin with the string `MonitorSelf`.

MyAddress: Description is not yet written.

MyCurrentTime: The time, represented as the number of second elapsed since the Unix epoch (00:00:00 UTC, Jan 1, 1970), at which the *condor_master* daemon last sent a ClassAd update to the *condor_collector*.

Name: The name of this resource; typically the same value as the `Machine` attribute, but could be customized by the site administrator. On SMP machines, the *condor_startd* will divide the CPUs up into separate slots, each with with a unique name. These names will be of the form "slot#@full.hostname", for example, "slot1@vulture.cs.wisc.edu", which signifies slot number 1 from vulture.cs.wisc.edu.

PublicNetworkIpAddr: Description is not yet written.

RealUid: The UID under which the *condor_master* is started.

UpdateSequenceNumber: An integer, starting at zero, and incremented with each ClassAd update sent to the *condor_collector*. The *condor_collector* uses this value to sequence the updates it receives.

Scheduler ClassAd Attributes

DaemonStartTime: The time that this daemon was started, represented as the number of second elapsed since the Unix epoch (00:00:00 UTC, Jan 1, 1970).

JobQueueBirthdate: Description is not yet written.

Machine: A string with the machine's fully qualified host name.

MaxJobsRunning: The same integer value as set by the evaluation of the configuration variable MAX_JOBS_RUNNING . See the definition at section 3.3.11 on page 208.

MonitorSelfAge: The number of seconds that this daemon has been running.

MonitorSelfCPUUsage: The fraction of recent CPU time utilized by this daemon.

MonitorSelfImageSize: The amount of virtual memory consumed by this daemon in Kbytes.

MonitorSelfRegisteredSocketCount: The current number of sockets registered by this daemon.

MonitorSelfResidentSetSize: The amount of resident memory used by this daemon in Kbytes.

MonitorSelfSecuritySessions: The number of open (cached) security sessions for this daemon.

MonitorSelfTime: The time, represented as the number of second elapsed since the Unix epoch (00:00:00 UTC, Jan 1, 1970), at which this daemon last checked and set the attributes with names that begin with the string MonitorSelf.

MyAddress: Description is not yet written.

MyCurrentTime: The time, represented as the number of second elapsed since the Unix epoch (00:00:00 UTC, Jan 1, 1970), at which the *condor_schedd* daemon last sent a ClassAd update to the *condor_collector*.

Name: The name of this resource; typically the same value as the Machine attribute, but could be customized by the site administrator. On SMP machines, the *condor_startd* will divide the CPUs up into separate slots, each with with a unique name. These names will be of the form "slot#@full.hostname", for example, "slot1@vulture.cs.wisc.edu", which signifies slot number 1 from vulture.cs.wisc.edu.

NumUsers: The integer number of distinct users with jobs in this *condor_schedd*'s queue.

PublicNetworkIpAddr: Description is not yet written.

QuillEnabled: The same boolean value as set in the configuration variable QUILL_ENABLED . See the definition at section 3.3.31 on page 262.

ScheddIpAddr: String with the IP and port address of the *condor_schedd* daemon which is publishing this Scheduler ClassAd.

ServerTime: Description is not yet written.

StartLocalUniverse: The same boolean value as set in the configuration variable START_LOCAL_UNIVERSE . See the definition at section 3.3.11 on page 207.

StartSchedulerUniverse: The same boolean value as set in the configuration variable START_SCHEDULER_UNIVERSE . See the definition at section 3.3.11 on page 208.

TotalFlockedJobs: The total number of jobs from this *condor_schedd* daemon that are currently flocked to other pools.

TotalHeldJobs: The total number of jobs from this *condor_schedd* daemon that are currently on hold.

TotalIdleJobs: The total number of jobs from this *condor_schedd* daemon that are currently idle.

TotalJobAds: The total number of all jobs (in all states) from this *condor_schedd* daemon.

TotalLocalIdleJobs: The total number of **local universe** jobs from this *condor_schedd* daemon that are currently idle.

TotalLocalRunningJobs: The total number of **local universe** jobs from this *condor_schedd* daemon that are currently running.

TotalRemovedJobs: The current number of all running jobs from this *condor_schedd* daemon that have remove requests.

TotalRunningJobs: The total number of jobs from this *condor_schedd* daemon that are currently running.

TotalSchedulerIdleJobs: The total number of **scheduler universe** jobs from this *condor_schedd* daemon that are currently idle.

TotalSchedulerRunningJobs: The total number of **scheduler universe** jobs from this *condor_schedd* daemon that are currently running.

UpdateSequenceNumber: An integer, starting at zero, and incremented with each ClassAd update sent to the *condor_collector*. The *condor_collector* uses this value to sequence the updates it receives.

VirtualMemory: Description is not yet written.

WantResAd: A boolean value that when True causes the *condor_negotiator* daemon to send to this *condor_schedd* daemon a full machine ClassAd corresponding to a matched job.

Negotiator ClassAd Attributes

DaemonStartTime: The time that this daemon was started, represented as the number of second elapsed since the Unix epoch (00:00:00 UTC, Jan 1, 1970).

LastNegotiationCycleActiveSubmitterCount<X>: The integer number of submitters the *condor_negotiator* attempted to negotiate with in the negotiation cycle. The number <X> appended to the attribute name indicates how many negotiation cycles ago this cycle happened.

LastNegotiationCycleDuration<X>: The number of seconds that it took to complete the negotiation cycle. The number <X> appended to the attribute name indicates how many negotiation cycles ago this cycle happened.

LastNegotiationCycleMatches<X>: The number of successful matches that were made in the negotiation cycle. The number <X> appended to the attribute name indicates how many negotiation cycles ago this cycle happened.

LastNegotiationCycleDuration<X>: The number of rejections that occurred in the negotiation cycle. The number <X> appended to the attribute name indicates how many negotiation cycles ago this cycle happened.

LastNegotiationCycleStartTime<X>: The time, represented as the number of second elapsed since the Unix epoch (00:00:00 UTC, Jan 1, 1970), at which the negotiation cycle started. The number <X> appended to the attribute name indicates how many negotiation cycles ago this cycle happened.

LastNegotiationCycleSubmittersFailed<X>: A string containing a space and comma-separated list of the names of all submitters who failed to negotiate in the negotiation cycle. One possible cause of failure is a communication timeout. This list does not include submitters who ran out of time due to NEGOTIATOR_MAX_TIME_PER_SUBMITTER . Those are listed separately in LastNegotiationCycleSubmittersOutOfTime<X>. The number <X> appended to the attribute name indicates how many negotiation cycles ago this cycle happened.

LastNegotiationCycleSubmittersOutOfTime<X>: A string containing a space and comma-separated list of the names of all submitters who ran out of time due to NEGOTIATOR_MAX_TIME_PER_SUBMITTER in the negotiation cycle. The number <X> appended to the attribute name indicates how many negotiation cycles ago this cycle happened.

Machine: A string with the machine's fully qualified host name.

MyAddress: Description is not yet written.

MyCurrentTime: The time, represented as the number of second elapsed since the Unix epoch (00:00:00 UTC, Jan 1, 1970), at which the *condor_schedd* daemon last sent a ClassAd update to the *condor_collector*.

Name: The name of this resource; typically the same value as the `Machine` attribute, but could be customized by the site administrator. On SMP machines, the *condor_startd* will divide the CPUs up into separate slots, each with with a unique name. These names will be of the form "slot#@full.hostname", for example, "slot1@vulture.cs.wisc.edu", which signifies slot number 1 from vulture.cs.wisc.edu.

NegotiatorIpAddr: String with the IP and port address of the *condor_negotiator* daemon which is publishing this Negotiator ClassAd.

PublicNetworkIpAddr: Description is not yet written.

UpdateSequenceNumber: An integer, starting at zero, and incremented with each ClassAd update sent to the *condor_collector*. The *condor_collector* uses this value to sequence the updates it receives.

Collector ClassAd Attributes

CollectorIpAddr: String with the IP and port address of the *condor_collector* daemon which is publishing this ClassAd.

CurrentJobsRunningAll: An integer value representing the sum of all jobs running under all universes.

CurrentJobsRunning<universe>: An integer value representing the current number of jobs running under the universe which forms the attribute name. For example

```
CurrentJobsRunningVanilla = 567
```

identifies that the *condor_collector* counts 567 vanilla universe jobs currently running. `<universe>` is one of `Unknown`, `Standard`, `Vanilla`, `Scheduler`, `Java`, `Parallel`, `VM`, or `Local`. There are other universes, but they are not listed here, as they represent ones that are no longer used in Condor.

DaemonStartTime: The time that this daemon was started, represented as the number of second elapsed since the Unix epoch (00:00:00 UTC, Jan 1, 1970).

HostsClaimed: Description is not yet written.

HostsOwner: Description is not yet written.

HostsTotal: Description is not yet written.

HostsUnclaimed: Description is not yet written.

IdleJobs: Description is not yet written.

Machine: A string with the machine's fully qualified host name.

MaxJobsRunning<universe: An integer value representing the sum of all `MaxJobsRunning<universe>` values.

MaxJobsRunning<universe>: An integer value representing largest number of currently running jobs ever seen under the universe which forms the attribute name, over the life of this *condor_collector* process. For example

```
MaxJobsRunningVanilla = 401
```

identifies that the *condor_collector* saw 401 vanilla universe jobs currently running at one point in time, and that was the largest number it had encountered. `<universe>` is one of `Unknown`, `Standard`, `Vanilla`, `Scheduler`, `Java`, `Parallel`, `VM`, or `Local`. There are other universes, but they are not listed here, as they represent ones that are no longer used in Condor.

MyAddress: Description is not yet written.

MyCurrentTime: The time, represented as the number of second elapsed since the Unix epoch (00:00:00 UTC, Jan 1, 1970), at which the *condor_schedd* daemon last sent a ClassAd update to the *condor_collector*.

Name: The name of this resource; typically the same value as the `Machine` attribute, but could be customized by the site administrator. On SMP machines, the *condor_startd* will divide the CPUs up into separate slots, each with with a unique name. These names will be of the form "slot#@full.hostname", for example, "slot1@vulture.cs.wisc.edu", which signifies slot number 1 from vulture.cs.wisc.edu.

RunningJobs: Description is not yet written.

UpdateInterval: Description is not yet written.

UpdateSequenceNumber: Description is not yet written.

UpdatesHistory: Description is not yet written.

UpdatesLost: Description is not yet written.

UpdatesSequenced: Description is not yet written.

UpdatesTotal: Description is not yet written.

ClassAd Attributes Added by the *condor_collector*

These attributes are only added if `COLLECTOR_DAEMON_STATS` is True. See page 227 for more information on configuration variable `COLLECTOR_DAEMON_STATS`.

LastHeardFrom: The time inserted into a daemon's ClassAd representing the time that this *condor_collector* last received a message from the daemon. Time is represented as the number of second elapsed since the Unix epoch (00:00:00 UTC, Jan 1, 1970).

UpdatesHistory: A bitmap representing the status of the most recent updates received from the daemon. This attribute is only added if `COLLECTOR_DAEMON_HISTORY_SIZE` is non-zero. See page 227 for more information on this setting.

UpdatesLost: An integer count of the number of updates from the daemon that were lost since the *condor_collector* started running.

UpdatesSequenced: An integer count of the number of updates received from the daemon for which the *condor_collector* can tell how many were lost, since the *condor_collector* started running.

UpdatesTotal: An integer count started when the *condor_collector* started running, representing the sum of the number of updates actually received from the daemon plus the number of updates that the *condor_collector* determined were lost.

CHAPTER

ELEVEN

Appendix B: Magic Numbers

Table 11.1: *condor_shadow* Exit Codes

Value	*Error Name*	*Description*
4	JOB_EXCEPTION	the job exited with an exception
44	DPRINTF_ERROR	there was a fatal error with dprintf()
100	JOB_EXITED	the job exited (not killed)
101	JOB_CKPTED	the job did produce a checkpoint
102	JOB_KILLED	the job was killed
103	JOB_COREDUMPED	the job was killed and a core file was produced
105	JOB_NO_MEM	not enough memory to start the condor_shadow
106	JOB_SHADOW_USAGE	incorrect arguments to condor_shadow
107	JOB_NOT_CKPTED	the job vacated without a checkpoint
107	JOB_SHOULD_REQUEUE	same number as JOB_NOT_CKPTED, to achieve the same behavior. This exit code implies that we want the job to be put back in the job queue and run again.
108	JOB_NOT_STARTED	can not connect to the *condor_startd* or request refused
109	JOB_BAD_STATUS	job status != RUNNING on start up
110	JOB_EXEC_FAILED	exec failed for some reason other than ENOMEM
111	JOB_NO_CKPT_FILE	there is no checkpoint file (as it was lost)
112	JOB_SHOULD_HOLD	the job should be put on hold
113	JOB_SHOULD_REMOVE	the job should be removed
114	JOB_MISSED_DEFERRAL_TIME	the job goes on hold, because it did not run within the specified window of time
115	JOB_EXITED_AND_CLAIM_CLOSING	the job exited (not killed) but the *condor_startd* is not accepting any more jobs on this claim

Table 11.2: User Log Event Codes

Event Code	*Description*
0	Submit
1	Execute
2	Executable error
3	Checkpointed
4	Job evicted
5	Job terminated
6	Image size
7	Shadow exception
8	Generic
9	Job aborted
10	Job suspended
11	Job unsuspended
12	Job held
13	Job released
14	Node execute
15	Node terminated
16	Post script terminated
17	Globus submit (no longer used)
18	Globus submit failed
19	Globus resource up (no longer used)
20	Globus resource down (no longer used)
21	Remote error
22	Job disconnected
23	Job reconnected
24	Job reconnect failed
25	Grid resource up
26	Grid resource down
27	Grid submit
28	Job ClassAd attribute values added to event log
29	Job status unknown
30	Job status known
31	Job stage in
32	Job stage out

Table 11.3: Well-Known Port Numbers

Server	*Port Number*
condor_negotiator	9614 (obsolete, now dynamically allocated)
condor_collector	9618
GT2 gatekeeper	2119
gridftp	2811
GT4 web services	8443
GCB local	65430
GCB public	65432

Table 11.4: DaemonCore Commands

Number	*Name*
60000	DC_RAISESIGNAL
60001	DC_PROCESSEXIT
60002	DC_CONFIG_PERSIST
60003	DC_CONFIG_RUNTIME
60004	DC_RECONFIG
60005	DC_OFF_GRACEFUL
60006	DC_OFF_FAST
60007	DC_CONFIG_VAL
60008	DC_CHILDALIVE
60009	DC_SERVICEWAITPIDS
60010	DC_AUTHENTICATE
60011	DC_NOP
60012	DC_RECONFIG_FULL
60013	DC_FETCH_LOG
60014	DC_INVALIDATE_KEY
60015	DC_OFF_PEACEFUL
60016	DC_SET_PEACEFUL_SHUTDOWN
60017	DC_TIME_OFFSET
60018	DC_PURGE_LOG

Table 11.5: DaemonCore Daemon Exit Codes

Exit Code	*Description*
0	Normal exit of daemon
99	`DAEMON_SHUTDOWN` evaluated to `True`

INDEX