Red Hat Enterprise MRG 1.3

Grid User Guide

Use and configuration information for MRG Grid

Lana Brindley

Scott Mumford

Red Hat Enterprise MRG 1.3 Grid User Guide
Use and configuration information for MRG Grid
Edition 7

Author	Lana Brindley	lbrindle@redhat.com
Author	Scott Mumford	smumford@redhat.com

1801 Varsity Drive
Raleigh, NC 27606-2072 USA
Phone: +1 919 754 3700
Phone: 888 733 4281
Fax: +1 919 754 3701

This book covers use and operation of the MRG Grid component of the Red Hat Enterprise MRG distributed computing platform. For installation instructions, see the *MRG Grid Installation Guide*.

Preface

Red Hat Enterprise MRG

This book contains information on the use and operation of the MRG Grid component of Red Hat Enterprise MRG. Red Hat Enterprise MRG is a high performance distributed computing platform consisting of three components:

1. *M*essaging — Cross platform, high performance, reliable messaging using the Advanced Message Queuing Protocol (AMQP) standard.
2. *R*ealtime — Consistent low-latency and predictable response times for applications that require microsecond latency.
3. *G*rid — Distributed High Throughput (HTC) and High Performance Computing (HPC).

All three components of Red Hat Enterprise MRG are designed to be used as part of the platform, but can also be used separately.

MRG Grid

Grid computing allows organizations to fully utilize their computing resources to complete high-performance tasks. By monitoring all resources - rack-mounted clusters and general workstations - for availability, any spare computing power can be redirected towards other, more intensive tasks until it is explicitly required again. This allows a standard networked system to operate in a way that is similar to a supercomputer.

MRG Grid provides high throughput computing and enables enterprises to achieve higher peak computing capacity as well as improved infrastructure utilization by leveraging their existing technology to build high performance grids. MRG Grid provides a job queueing mechanism, scheduling policy, priority scheme, resource monitoring, and resource management. Users submit their jobs to MRG Grid, where they are placed into a queue. MRG Grid then chooses when and where to run the jobs based upon a policy, carefully monitors their progress, and ultimately informs the user upon completion.

MRG Grid is based on the *Condor Project*[1] developed within the *University of Wisconsin-Madison*[2]. Condor also offers a comprehensive library of freely available documentation in its *Manual*[3].

1. Document Conventions

This manual uses several conventions to highlight certain words and phrases and draw attention to specific pieces of information.

In PDF and paper editions, this manual uses typefaces drawn from the *Liberation Fonts*[4] set. The Liberation Fonts set is also used in HTML editions if the set is installed on your system. If not, alternative but equivalent typefaces are displayed. Note: Red Hat Enterprise Linux 5 and later includes the Liberation Fonts set by default.

[1] http://www.cs.wisc.edu/condor/
[2] http://www.wisc.edu/
[3] http://www.cs.wisc.edu/condor/manual/
[4] https://fedorahosted.org/liberation-fonts/

1.1. Typographic Conventions

Four typographic conventions are used to call attention to specific words and phrases. These conventions, and the circumstances they apply to, are as follows.

`Mono-spaced Bold`

Used to highlight system input, including shell commands, file names and paths. Also used to highlight keycaps and key combinations. For example:

> To see the contents of the file **`my_next_bestselling_novel`** in your current working directory, enter the **`cat my_next_bestselling_novel`** command at the shell prompt and press **`Enter`** to execute the command.

The above includes a file name, a shell command and a keycap, all presented in mono-spaced bold and all distinguishable thanks to context.

Key combinations can be distinguished from keycaps by the hyphen connecting each part of a key combination. For example:

> Press **`Enter`** to execute the command.
>
> Press **`Ctrl`**+**`Alt`**+**`F1`** to switch to the first virtual terminal. Press **`Ctrl`**+**`Alt`**+**`F7`** to return to your X-Windows session.

The first paragraph highlights the particular keycap to press. The second highlights two key combinations (each a set of three keycaps with each set pressed simultaneously).

If source code is discussed, class names, methods, functions, variable names and returned values mentioned within a paragraph will be presented as above, in **`mono-spaced bold`**. For example:

> File-related classes include **`filesystem`** for file systems, **`file`** for files, and **`dir`** for directories. Each class has its own associated set of permissions.

Proportional Bold

This denotes words or phrases encountered on a system, including application names; dialog box text; labeled buttons; check-box and radio button labels; menu titles and sub-menu titles. For example:

> Choose **System** → **Preferences** → **Mouse** from the main menu bar to launch **Mouse Preferences**. In the **Buttons** tab, click the **Left-handed mouse** check box and click **Close** to switch the primary mouse button from the left to the right (making the mouse suitable for use in the left hand).
>
> To insert a special character into a **gedit** file, choose **Applications** → **Accessories** → **Character Map** from the main menu bar. Next, choose **Search** → **Find…** from the **Character Map** menu bar, type the name of the character in the **Search** field and click **Next**. The character you sought will be highlighted in the **Character Table**. Double-click this highlighted character to place it in the **Text to copy** field and then click the **Copy** button. Now switch back to your document and choose **Edit** → **Paste** from the **gedit** menu bar.

The above text includes application names; system-wide menu names and items; application-specific menu names; and buttons and text found within a GUI interface, all presented in proportional bold and all distinguishable by context.

`Mono-spaced Bold Italic` or ***`Proportional Bold Italic`***

Whether mono-spaced bold or proportional bold, the addition of italics indicates replaceable or variable text. Italics denotes text you do not input literally or displayed text that changes depending on circumstance. For example:

> To connect to a remote machine using ssh, type **`ssh`** ***`username@domain.name`*** at a shell prompt. If the remote machine is **`example.com`** and your username on that machine is john, type **`ssh john@example.com`**.
>
> The **`mount -o remount`** ***`file-system`*** command remounts the named file system. For example, to remount the **`/home`** file system, the command is **`mount -o remount /home`**.
>
> To see the version of a currently installed package, use the **`rpm -q`** ***`package`*** command. It will return a result as follows: ***`package-version-release`***.

Note the words in bold italics above — username, domain.name, file-system, package, version and release. Each word is a placeholder, either for text you enter when issuing a command or for text displayed by the system.

Aside from standard usage for presenting the title of a work, italics denotes the first use of a new and important term. For example:

> Publican is a *DocBook* publishing system.

1.2. Pull-quote Conventions

Terminal output and source code listings are set off visually from the surrounding text.

Output sent to a terminal is set in **`mono-spaced roman`** and presented thus:

```
books        Desktop   documentation  drafts  mss    photos   stuff  svn
books_tests  Desktop1  downloads      images  notes  scripts  svgs
```

Source-code listings are also set in **`mono-spaced roman`** but add syntax highlighting as follows:

```
package org.jboss.book.jca.ex1;

import javax.naming.InitialContext;

public class ExClient
{
   public static void main(String args[])
       throws Exception
   {
      InitialContext iniCtx = new InitialContext();
      Object         ref    = iniCtx.lookup("EchoBean");
      EchoHome       home   = (EchoHome) ref;
      Echo           echo   = home.create();

      System.out.println("Created Echo");

      System.out.println("Echo.echo('Hello') = " + echo.echo("Hello"));
   }
}
```

1.3. Notes and Warnings

Finally, we use three visual styles to draw attention to information that might otherwise be overlooked.

Note

Notes are tips, shortcuts or alternative approaches to the task at hand. Ignoring a note should have no negative consequences, but you might miss out on a trick that makes your life easier.

Important

Important boxes detail things that are easily missed: configuration changes that only apply to the current session, or services that need restarting before an update will apply. Ignoring a box labeled 'Important' won't cause data loss but may cause irritation and frustration.

Warning

Warnings should not be ignored. Ignoring warnings will most likely cause data loss.

2. Getting Help and Giving Feedback

2.1. Do You Need Help?

If you experience difficulty with a procedure described in this documentation, you can find help in the following ways:

Red Hat Knowledgebase
: Visit the Red Hat Knowledgebase at *http://kbase.redhat.com* to search or browse through technical support articles about Red Hat products.

Red Hat Global Support Services
: Your Red Hat subscription entitles you to support from Red Hat Global Support Services (GSS). Visit *http://support.redhat.com* for more information about obtaining help from GSS.

Other Red Hat documentation
: Access other Red Hat documentation at *http://www.redhat.com/docs*

Red Hat electronic mailing lists
: Red Hat hosts a large number of electronic mailing lists for discussion of Red Hat software and technology. You can find a list of publicly available lists at *https://www.redhat.com/mailman/listinfo*. Click on the name of any list to subscribe to that list or to access the list archives.

2.2. We Need Feedback!

If you find a typographical error in this manual, or if you have thought of a way to make this manual better, we would love to hear from you! Please submit a report in Bugzilla: *http://bugzilla.redhat.com/* against the product **Red Hat Enterprise MRG.**

When submitting a bug report, be sure to mention the manual's identifier: *Grid_User_Guide*

If you have a suggestion for improving the documentation, try to be as specific as possible when describing it. If you have found an error, please include the section number and some of the surrounding text so we can find it easily.

Overview

MRG Grid uses the computational ability of many computers connected over a network to complete large or resource-intensive operations. MRG Grid harnesses existing resources by detecting when a workstation becomes idle, and then relinquishing that resource when it is required by another user. When a job is submitted to MRG Grid, it finds an idle machine on the network and begins running the job on that machine. There is no requirement for machines to share file systems, so machines across an entire enterprise can run a job, including machines in different administrative domains.

MRG Grid uses ClassAds to simplify job matching and submission. All machines in the MRG Grid pool advertise their resources, such as available RAM memory, CPU speed, and virtual memory size in a resource offer ClassAd. When a job is submitted, the submitter specifies a required and a desired set of properties, in a resource request ClassAd. MRG Grid matches and ranks resource offer ads with resource request ads, making certain that all requirements in both ads are satisfied. During this match-making process, MRG Grid 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 ClassAds over others.

Groups of researchers, engineers, and scientists have used MRG Grid to establish pools ranging in size from a handful to tens of thousands of workstations.

Configuration

This chapter describes how configuration is handled throughout the MRG Grid system, and explains how to change the configuration. A list of the configuration parameters, and examples of the default configuration files can be found in *Appendix A, Configuration options*.

Configuration in MRG Grid is determined by information held in multiple files across the entire system. The global configuration file is read first, and if it does not exist, MRG Grid will not run. The settings in the global configuration file are then extended or overwritten by configuration files located in the local configuration directory or other locations specified by the local configuration file parameter.

As MRG Grid reads each consecutive configuration file, the information is layered upon the details it already knows. If any parameters are repeated, the information read last overwrites the information read earlier.

Configuration parameters are specified using key-value pairs separated by an equals (=) sign. There must be a space character on each side of the equals sign. Valid configuration parameters look like this:

```

name = value
```

The different configuration files are parsed in the following order:

1. *Global configuration file*

   A global configuration file is required by all machines in the pool and should be the same across all nodes. This file should not be directly edited. The file is managed by the **condor.rpm** package. An example of what the global configuration file looks like is in *Example A.1, “The default global configuration file”*.

   MRG Grid will look in different places for the global configuration file, in the following order:

   a. The filename specified in the **CONDOR_CONFIG** environment variable

   b. **/etc/condor/condor_config**

      MRG Grid places the global configuration file here by default

   c. **/usr/local/etc/condor_config**

   d. **~condor/condor_config**

   Note

   If a file is specified in the **CONDOR_CONFIG** environment variable and there is a problem reading that file, MRG Grid will print an error message and exit. It will not continue to search the other options. Leaving the **CONDOR_CONFIG** environment variable blank will ensure that MRG Grid will search through the other options.

   If a valid configuration file is not found in any of the searched locations, MRG Grid will print an error message and exit.

2. *Local configuration directory*

The local configuration directory is located at **`/etc/condor/config.d`**. This location is defined by the default global configuration file. The local configuration directory provides an easy way to extend the configuration of MRG Grid by placing files that contain configuration parameters inside the directory.

Files in this directory will override settings in the global file for that machine. Files must contain configuration parameters. Files in this directory are read in lexigraphical order. If there are duplicate parameters in the files, parameters that are read later will override those values read earlier.

To ensure the files are ordered correctly, each filename is preceded with a two-digit number, using the following ranges:

- *`00`* - personal condor (included by default)
- *`10-40`* - user configuration files

  Use this range to extend the configuration of MRG Grid
- *`50-80`* - MRG Grid package configuration files
- *`99`* - Reserved for the remote configuration feature

3. *Local configuration file*

   The **`LOCAL_CONFIG_FILE`** parameter in the global configuration file can be used to specify the location of files with configuration to be read.

   Note
   The **`LOCAL_CONFIG_FILE`** parameter is used by the remote configuration feature. Do not set this parameter if remote configuration is used.

4. *Local configuration directory*

   If the local configuration directory has been changed by a configuration setting, it will be read a second time. Only files added since it was last read will be processed.

Warning
Many text editors create backup files, identified by a tilde (~) after the filename. MRG Grid cannot differentiate between these backup files and ordinary configuration files. Leaving out-of-date backup files could result in configuration settings being overridden. Always delete editor backup files once they are no longer required.

Creating a user configuration file

1. Switch to the root user, and create a file in the **`/etc/condor/config.d`** directory:

```
# touch /etc/condor/config.d/10myconfigurationfile
```

2. Open the new file using your preferred text editor, and add or edit the configuration parameters as required

3. Save the file

4. Restart the **condor** service:

```
# service condor restart
Stopping condor services:                [  OK  ]
Starting condor services:                [  OK  ]
```

Once MRG Grid has completed parsing the four configuration file locations, it will check for environment variables. These configuration variables are case-insensitive, and are prefixed by either **_CONDOR_** or **_condor_**. MRG Grid parses environment variables last, subsequently any settings made this way will override conflicting settings in the configuration files.

Adding entries to configuration files

1. All entries in a configuration file use the same syntax. The entries are in the form:

   ```
   # This is a comment
   SUBSYSTEM_NAME.CONFIG_VARIABLE = VALUE
   ```

   Things to note about the syntax:

   - Each valid entry requires an operator of **=**
   - A line prefixed by a **#** symbol will be treated as a comment and ignored. The **#** symbol can only appear at the beginning of a line. It will not create a comment if it is used in the middle of a line.
   - The *SUBSYSTEM_NAME* is optional
   - There must be a space character on either side of the **=** sign

2. An entry can continue over multiple lines by placing a \ character at the end of the line to be continued. For example:

   ```
   ADMIN_MACHINES = condor.example.com, raven.example.com, \
   stork.example.com, ostrich.example.com \
   bigbird.example.com
   ```

   Important

   The line continuation character will also work within a comment, which will cause MRG Grid to ignore the second line. The following example would be ignored entirely:

   ```
   # This comment has line continuation \
   characters, so FOO will not be set \
   FOO = BAR
   ```

Initial configuration

Review the configuration files stored at **`/etc/condor/condor_config`** and **`/etc/condor/config.d/00personal_condor.config`** before starting MRG Grid.

The default configuration sets up a **`Personal Condor`**. **`Personal Condor`** is a specific style of installation suited for individual users who do not have their own pool of machines. To allow other machines to join the pool, specify the **`ALLOW_WRITE`** option in the local configuration directory.

1. To extend a Personal Condor installation and allow other multiple nodes, begin by creating a file in **`/etc/condor/config.d`**:

   ```
   # touch /etc/condor/config.d/10pool_access
   ```

2. Open the new file in your preferred text editor. A value for the **`ALLOW_WRITE`** configuration parameter must be specified in order to allow machines to join your pool and submit jobs. Any machine that you give write access to using the **`ALLOW_WRITE`** option should also be given read access using the **`ALLOW_READ`** option:

   ```
   ALLOW_WRITE = *.your.domain.com
   ```

   Warning

   The simplest option is to include **`ALLOW_WRITE = *`** in the configuration file. However, this will allow anyone to submit jobs or add machines to your pool. This is a serious security risk and therefore not recommended.

Executing a program to produce configuration entries

1. MRG Grid can run a specialized program to obtain configuration entries. To run a program from the configuration file, insert a **`|`** character at the end of the line. This syntax will only work with the configuration variable **`LOCAL_CONFIG_FILE`**. For example, to run a program located at **`/bin/make_the_config`**, use the following entry:

   ```
   LOCAL_CONFIG_FILE = /bin/make_the_config|
   ```

   The program **`/bin/make_the_config`** must output the configuration parameters on standard output for the configuration parameters to be included in the configuration.

2.1. Configuring MRG Grid for Microsoft Windows

This section contains information for using MRG Grid on systems running:

- Microsoft Windows XP Service Pack 3
- Microsoft Windows Server 2003

Under Microsoft Windows, the following features are supported:

- A graphical installation and setup program, which can perform a full installation. Information specified by the user in the setup program is stored in the system registry. The setup program can also update a current installation with a new release
- The ability to submit, run, and manage queues of jobs on a cluster of machines running Microsoft Windows
- All tools, including **`condor_q`**, **`condor_status`**, and **`condor_userprio`**
- The ability to customize job policies with ClassAds. The machine ClassAds contain all the information included in the Linux version, including current load average, RAM and virtual memory sizes, integer and floating-point performance, and keyboard and mouse idle times. Likewise, job ClassAds contain all information including system dependent entries such as dynamic updates of the job's image size and CPU usage
- Security mechanisms
- Support for SMP machines
- Jobs can be run at a lower operating system priority level. Jobs can be suspended, soft-killed with a **`WM_CLOSE`** message, or hard-killed automatically based upon policy expressions. For example, a job can be suspended whenever keyboard or 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
- Jobs that create multiple processes are accurately handled. For example, if a job spawns multiple processes and Condor needs to kill the job, all processes created by the job will also be terminated
- In addition to interactive tools, users and administrators can receive information by e-mail (standard SMTP) or by log files
- Provide job access to the running user's registry hive

The following features are not yet supported under Microsoft Windows operating systems:

- Accessing files via a network share that requires a kerberos ticket such as AFS is not supported
- The **`run_as_owner`** feature is disabled

Note

For information about installing MRG Grid on Microsoft Windows, see the *MRG Grid Installation Guide*.

Executing jobs with the user profile loaded

When Condor is running on dedicated run accounts, it can be configured to load the current account profile. The profile includes a set of personal directories and a registry hive, which are loaded under **`HKEY_CURRENT_USER`**. This can be useful if the job requires direct access to the user's registry entries. It can also be useful when the job requires an application that needs registry access.

This feature is enabled on the **`condor_startd`**, but only operates with the dedicated run account. For security reasons, the profiles are removed after the job has completed and exited. This ensures that any malicious jobs cannot discover the details of any previous jobs, or sabotage the registry for future jobs. It also ensures that the next job has a fresh registry hive.

To run a job with the current account profile, add the following line to the job's submit description file:

```
load_profile = True
```

Using scripts as job executables

It is possible to use scripts to run condor jobs on Microsoft Windows. Condor uses the file name's extension to determine how to handle the script. If the file name does not have an extension, it is assumed to be a Windows executable (**`.exe`**) file.

This feature can be used without changing the basic configuration, when using Perl scripts with **`ActivePerl`**. It is also possible to use Windows Scripting Host scripts, although some configuration changes are neccessary.

A registry lookup is performed to ensure that the correct interpreter is invoked, with the correct command line arguments for the scripting language. The configuration specifies values to be used in the registry lookup. Actions that can be used on a file (like **Open**, **Print**, and **Edit**) are referred to as *verbs*. They are specified in the configuration file, and invoked from the **`HKEY_CLASSES_ROOT`** registry hive.

Registry lookups use the following format:

```
HKEY_CLASSES_ROOT\FileType\Shell\OpenVerb\Command
```

`OpenVerb` identifies the verb. This is set in the Condor configuratin file, and aids the registry lookup.

`FileType` is the name of a file type, and is obtained from the file name extension. 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. In the following example, the *`Open2`* verb is specified for a Windows Scripting Host registry lookup 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 the above example, the *`Open2`* verb has been specified instead of the default *`Open`* verb for several scripts, including Windows Scripting Host scripts (using the extension **`.wsh`**). The *`Open2`* verb in Windows Scripting Host scripts allows standard input, standard output, and standard error to be redirected as needed for Condor jobs.

Allowing access to the user's registry

A common difficulty is encountered when a script interpreter requires access to the user's registry entries. 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 reads:

```
CScript Error: Loading your settings failed. (Access is denied.)
```

This error is resolved by giving explicit access to the submitting user's registry hive. Access can be allowed by using the **`load_profile`** command in the job's submit description file:

```
load_profile = True
```

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.

Starting and stopping jobs under Microsoft Windows

1. When Condor is about to start a job, the **`condor_startd`** service on the execute machine spawns a **`condor_starter`** process (referred to as the *starter*). The starter then creates:

   - A run account on the machine. The account is given the login name *`condor-reuse-slotX`*, where *X* is the slot number of the starter. This account is added to the group *`Users`*.

   - A temporary working directory for the job on the execute machine. The directory is named **`dir_XXX`**, where *XXX* is the process ID of the starter. The directory is created in the **`$(EXECUTE)`** directory as specified in the configuration file. Condor then grants write permission to this directory for the user account newly created for the job.

   - A new, non-visible WindowStation and desktop for the job. Permissions are set so that only the account that will run the job has access rights to the desktop. Any windows created by this job are not seen by anyone; the job is run in the background. To force the job to use the default desktop instead of creating a new one, set **`USE_VISIBLE_DESKTOP = True`** in the job submit file.

2. The starter then contacts the **`condor_shadow`** service (referred to as the *shadow*), which is running on the submitting machine, and copies the job's executable and input files. These files are placed into the temporary working directory for the job.

3. Once all the executable and input files have been received, the starter spawns the user's executable file. It changes the current working directory to the temporary working directory (that is, **`$(EXECUTE)/dir_XXX`**, where *XXX* is the process ID of the starter).

4. While the job is running, the starter monitors the CPU usage and image size of all processes started by the job. This information is sent to the shadow every five minutes, along with the total size of all files contained in the job's temporary working directory. The shadow then inserts this information into the job's ClassAd so that policy and scheduling expressions can make use of the information.

   The frequency of this check can be adjusted by changing the value (in seconds) of the **`STARTER_UPDATE_INTERVAL`** configuration parameter.

5. If the job completes successfully, the starter will terminate any processes started by the job which are still running. The starter searches the job's temporary working directory for any files which have been created or modified. Any files that are found are sent back to the shadow running on

the submit machine. The shadow then moves the files into the initial directory specified in the submit description file. If no initial directory was specified, the files are moved to the directory from which the user invoked the **condor_submit** command. Once all the output files are safely transferred back, the job is removed from the queue.

If the **condor_startd** is forced to kill the job before all output files are transferred, the job is not removed from the queue but is instead transitioned back to the *Idle* state.

If the **condor_startd** vacates 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 Microsoft Windows, since this method allows the job to clean up properly and free any resources that have been allocated.

When a job exits, the starter cleans up any processes left behind. If **when_to_transfer_output** is set to the default *ON_EXIT* in the submit description file, the job will switch states from *Running* to *Idle*, and no files will be transferred back. However, if it is set to *ALWAYS*, any files in the job's temporary working directory which were changed or modified will be sent back to the submitting machine. The shadow will put the files into a subdirectory under the **SPOOL** directory on the submitting machine. The job is then switched back to the *Idle* state until a different machine is found on which it can be run. When the job is restarted, Condor puts the executable and input files into the temporary working directory as before, as well as any files stored in the submit machine's **SPOOL** directory for that job.

By default, when a **WM_CLOSE** message is sent, the process receiving the message will exit. In some cases, the job can be coded to ignore it and not exit, but in this instance eventually the **condor_startd** will hard kill the job (if that is the policy desired by the administrator).

Note

If special cleanup work needs to occur when the job is being vacated, the Win32 **SetConsoleCtrlHandler()** function can be used to intercept a **WM_CLOSE** message. A **WM_CLOSE** message generates a **CTRL_CLOSE_EVENT**. See **SetConsoleCtrlHandler()** in the Win32 documentation for more information.

6. After the job has finished and any files have been transferred back, the starter deletes the temporary working directory, the temporary account (if one was created), the WindowStation, and the desktop. The starter will then exit. If the starter terminates abnormally, the **condor_startd** will attempt to clean up. If for some reason the **condor_startd** should disappear as well (which is only likely to happen if the machine is suddenly rebooted), the **condor_startd** will clean up once Condor has been restarted.

Security under Microsoft Windows

By default, the execute machine runs user jobs with the access token of an account dynamically created by Condor. This account has limited access rights and privileges. For example, in a situation where only administrator accounts have write access to **C:\WINNT**, then no Condor job that is run on that machine would be able to write to that location. The only files jobs can access on the execute machine are files accessible by the *Users* and *Everyone* groups, and files within the job's own temporary working directory.

Important

Condor for Microsoft Windows implements all the security mechanisms described in *Chapter 3, Security*.

Using a network file server

MRG Grid can be used with a network file server. The current version, however, cannot be **run_as_owner**. This section outlines several ways to use Condor with networked files.

Problems can arise when using Condor with a network file server. When a temporary user is created to run jobs, the file server will not allow it access to the files, as it has not been properly authenticated. There are several methods that can be used to work around this issue:

1. Access the file server as a different user, with a **net use** command and a login and password.

   For example, to copy a file from a server and then execute it:

   ```
   @echo off
   net use \\myserver\someshare MYPASSWORD /USER:MYLOGIN
   copy \\myserver\someshare\my-program.exe
   my-program.exe
   ```

   This method simply authenticates to the file server with a login other than the temporary Condor login. The disadvantage with this method is that the password is stored and transferred as clear text, which could be a potential security issue.

2. Access the file server as guest.

   For example, to copy a file from a server and then execute it:

   ```
   @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. If the **GUEST** account is enabled on the server, the user will be authenticated to the server as user **GUEST**. Set the access control lists (ACLs) so that the **GUEST** user or the **EVERYONE** group has access to the share **someshare** and the directories and files there. The disadvantage of this method is that the **GUEST** account must be enabled on the file server.

   Warning

   This method should be used with extreme caution. Ensure the file server is well protected behind a firewall that blocks SMB traffic.

3. Access the file server with a **NULL** security descriptor.

   This method allows shared files to be specified by adding them to the registry. Once this is set up, a batch file wrapper can then be used:

```
net use z: \\myserver\someshare /USER:""
z:\my-program.exe
```

In the above example, **`someshare`** is in the list of allowed **`NULL`** session shares. To edit the list, run **`regedit.exe`** and navigate to this key:

```
HKEY_LOCAL_MACHINE\
   SYSTEM\
     CurrentControlSet\
       Services\
         LanmanServer\
           Parameters\
             NullSessionShares
```

The key can then be edited. This key only accepts binary values, so the share must be input using the hex ASCII codes. Each share is separated by a null (*`0x00`*) and the final entry in the list is terminated with two nulls.

This method is slightly more complex to set up, but it provides a relatively safe way to have one partly-public share without needing to open the whole guest account. The shares that can be accessed can be directly specified using the registry value mentioned above.

Security

Security in MRG Grid is implemented by ensuring every communication is subject to various checks. When condor services communicate with each other, authentication is required to ensure that the system has not been compromised. User requests are checked to ensure the user account has the appropriate privileges before the request is acted upon.

Communication occurs between a client and a service. The client initiates the request, and a service processes the command and responds. Tools such as **`condor_submit`** and **`condor_config`** are always clients, sending requests to services. Services can also interact with each other, where the daemon making the request acts as the client.

3.1. Security Contexts

All requests are categorized into contexts. In order to make a request, authorization must be granted at the appropriate level.

`READ`
: Able to obtain or read information. *`READ`* access is required to view the status of the pool using **`condor_status`**; check a job queue with **`condor_q`**; view user priorities with **`condor_userprio`**. *`READ`* access will not allow changes to be made, and it will not allow jobs to be submitted.

`WRITE`
: Able to send or write information. *`WRITE`* access is required for submitting jobs using **`condor_submit`**; advertising a machine so it appears in the pool, which is usually done automatically by the **`condor_startd`** service. The *`WRITE`* context implies *`READ`* access.

`ADMINISTRATOR`
: Provides additional administrator rights. *`ADMINISTRATOR`* rights are required to change user priorities using **`condor_userprio -set`**; turn Condor on and off using **`condor_on`** and **`condor_off`**. The *`ADMINISTRATOR`* context implies both *`READ`* and *`WRITE`* access.

`SOAP`
: Provides access to the Web Services (SOAP) interface. *`SOAP`* is not a general context, and should not be used with configuration variables for authentication, encryption, and integrity checks.

`CONFIG`
: Provides access to modify the configuration of services using **`condor_config_val`**. By default, this level of access can change any configuration parameters of a Condor pool. The *`CONFIG`* context implies *`READ`* access.

`OWNER`
: Provides a context suitable for the owner of a machine to use. The *`OWNER`* context can be used to perform commands such as the **`condor_vacate`** command, which causes the **`condor_startd`** to vacate any job currently running on a machine.

`DAEMON`
: Provides access to internal commands. An internal command is communication that occurs between services, such as the **`condor_startd`** sending ClassAd updates to the **`condor_collector`**. This context is only required for the user account that runs the Condor

services. The *`DAEMON`* context implies both *`READ`* and *`WRITE`* access. Any configuration setting for this context that is not defined will default to the corresponding setting for the *`WRITE`* context.

`NEGOTIATOR`
: Used explicitly to verify commands sent by the **`condor_negotiator`** service, which runs on the central manager. Commands requiring this context are those that instruct **`condor_schedd`** to begin negotiating, and those that tell an available **`condor_startd`** that it has been matched to a **`condor_schedd`** with jobs to run. The *`NEGOTIATOR`* level of access implies *`READ`* access.

`ADVERTISE_MASTER`
: Used explicitly for tasks used to advertise a **`condor_master`** to the collector. Any configuration setting for this context that is not defined will default to the corresponding setting for the *`DAEMON`* context.

`ADVERTISE_STARTD`
: Used explicitly for tasks used to advertise a **`condor_startd`** to the collector. Any configuration setting for this context that is not defined will default to the corresponding setting for the *`DAEMON`* context.

`ADVERTISE_SCHEDD`
: Used explicitly for tasks used to advertise a **`condor_schedd`** to the collector. Any configuration setting for this context that is not defined will default to the corresponding setting for the *`DAEMON`* context.

`CLIENT`
: This context is used only for internal communications when services contact other services. Unlike the other access policies, it provides access control for the client initiating the operation, instead of the server that is contacted.

This table provides a list of commands that will be accepted by each service, and the security context required for that command to be accepted.

Activity	Service	Security Context
Reconfigure a service with **`condor_reconfig`**	All services	*`WRITE`*
Signalling	All services	*`DAEMON`*
Keep alives	All services	*`DAEMON`*
Read configuration	All services	*`READ`*
Runtime configuration	All services	*`ALLOW`*
Daemon off	All services	*`ADMINISTRATOR`*
Fetch or purge logs	All services	*`ADMINISTRATOR`*
Activate, request, or release a claim.	**`condor_startd`**	*`WRITE`*
Retrieve startd or job information with **`condor_preen`**	**`condor_startd`**	*`READ`*
Heartbeat	**`condor_startd`**	*`DAEMON`*
Deactivate claim	**`condor_startd`**	*`DAEMON`*

Activity	Service	Security Context
condor_vacate Used to stop running jobs	**condor_startd**	*OWNER*
Retrieve negotiation information	**condor_startd**	*NEGOTIATOR*
ClassAd commands	**condor_startd**	*WRITE*
VM Universe commands	**condor_startd**	*DAEMON*
ClassAd commands	**condor_starter**	*WRITE*
Hold jobs	**condor_starter**	*DAEMON*
Create job security session	**condor_starter**	*DAEMON*
Start SSHD	**condor_starter**	*READ*
Initiate a new negotiation cycle	**condor_negotiator**	*WRITE*
Retrieve the current user priorities with **userprio**	**condor_negotiator**	*READ*
Set user priorities with **userprio -set**	**condor_negotiator**	*ADMINISTRATOR*
Reschedule	**condor_negotiator**	*DAEMON*
Reset usage	**condor_negotiator**	*ADMINISTRATOR*
Delete user	**condor_negotiator**	*ADMINISTRATOR*
Set usage statistics	**condor_negotiator**	*ADMINISTRATOR*
Update the **condor_collector** with new **condor_master** ClassAds	**condor_collector**	*ADVERTISE_MASTER*
Update the **condor_collector** with new **condor_schedd** ClassAds	**condor_collector**	*ADVERTISE_SCHEDD*
Commands that update the **condor_collector** with new **condor_startd** ClassAds	**condor_collector**	*ADVERTISE_STARTD*
All other commands that update the **condor_collector** with new ClassAds	**condor_collector**	*DAEMON*
All commands that query the **condor_collector** for ClassAds	**condor_collector**	*READ*
Query the collector	**condor_collector**	*ADMINISTRATOR*
Invalidate all AdTypes except **STARTD**, **SCHEDD**, or **MASTER**	**condor_collector**	*DAEMON*
Update the collector	**condor_collector**	*ALLOW*
Merge **STARTD**	**condor_collector**	*NEGOTIATOR*
Begin negotiating to match jobs	**condor_schedd**	*NEGOTIATOR*

Activity	Service	Security Context
Get matches	**condor_schedd**	*DAEMON*
Begin negotiation cycle with **condor_reschedule**	**condor_schedd**	*WRITE*
View the status of the job queue	**condor_schedd**	*READ*
Reconfigure	**condor_schedd**	*OWNER*
File operations. Release claim, kill job, or spool job.	**condor_schedd**	*WRITE*
Reuse shadow	**condor_schedd**	*DAEMON*
Update shadow	**condor_schedd**	*DAEMON*
Startd heartbeat	**condor_schedd**	*DAEMON*
Store credentials	**condor_schedd**	*WRITE*
Write to the job queue	**condor_schedd**	*WRITE*
Transfer the job queue	**condor_schedd**	*WRITE*
All commands	**condor_master**	*ADMINISTRATOR*
All high availability functionality	**condor_had**	*DAEMON*

Table 3.1. Registered Commands

3.2. Security negotiation

Security settings are determined through a security negotiation process. Security negotiation is used to determine what security features are required for each connection: authentication, encryption, integrity checking, or a combination. Negotiation also defines which protocol to use for each feature.

Setting security configuration parameters

1. Configuration parameters are used during the security negotiation process to determine which feature and protocols are to be used. All security configuration parameters follow the format:

```
SEC_CONTEXT_FEATURE = VALUE
```

2. Using the above syntax, specify the feature against which the policy is to be set. The feature can be any one of:

   - *AUTHENTICATION*
   - *ENCRYPTION*
   - *INTEGRITY*
   - *NEGOTIATION*

3. Specify the context for the policy. Context can be any one of:

   - *CLIENT*

- *READ*
- *WRITE*
- *ADMINISTRATOR*
- *CONFIG*
- *OWNER*
- *DAEMON*
- *NEGOTIATOR*
- *ADVERTISE_MASTER*
- *ADVERTISE_STARTD*
- *ADVERTISE_SCHEDD*
- *DEFAULT*

  The *DEFAULT* value provides a way to set a policy for all access levels that do not have a specific configuration variable defined.

4. Specify a value for the policy. The value can be any one of:

- *REQUIRED*
- *PREFERRED*
- *OPTIONAL*
- *NEVER*

Encryption and integrity checks can only be enabled if authentication can occur. The authentication process provides a key exchange, which necessary for these features.

To determine the policy for all outgoing commands, set a policy of:

```
SEC_CLIENT_FEATURE
```

Setting a policy for incoming commands requires setting context. It is good policy in most situations to require authentication for all incoming administrative requests, while enquiries on the status of a pool do not need to be so restrictive. In order to implement this, set the following policies on the server:

```
SEC_ADMINISTRATOR_AUTHENTICATION = REQUIRED
SEC_READ_AUTHENTICATION          = OPTIONAL
```

This example demonstrates how to set security negotiation policies, to create a security environment that requires as little authentication as possible.

On the job submission machine, set the following security policy:

```
SEC_DEFAULT_AUTHENTICATION = OPTIONAL
```

However, if the machine running the **condor_schedd** requires authentication to be set, it will have had this policy specified:

```
SEC_DEFAULT_AUTHENTICATION = REQUIRED
```

In this case, submitted jobs will still be accepted. This is because the security negotiation process will enforce the most restrictive security policy. Some commands, such as **condor_submit**, always require authentication, regardless of the specified policy. Other commands, such as **condor_q**, do not always require authentication. In this example, the server's policy would force any **condor_q** queries to be authenticated, where a different policy could allow **condor_q** to occur without authentication.

Example 3.1. Setting security negotiation policies

3.3. Authentication

On the client side of a communication, one of the following configuration parameters are used to specify whether or not authentication should occur:

```
SEC_DEFAULT_AUTHENTICATION
SEC_CLIENT_AUTHENTICATION
```

For services, there are a larger number of configuration parameters used to specify whether authentication should occur, based upon the necessary context:

```
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
```

If no variable is defined for **SEC_*access-level*_AUTHENTICATION**, a default value of *OPTIONAL* will be used. In this case, authentication is required for any operation which modifies the job queue, such as **condor_qedit** and **condor_rm**.

If no variable is defined for **SEC_*access-level*_AUTHENTICATION_METHODS**, a default value of *FS, KERBEROS* will be defined. On a Microsoft Windows machine, it will default to *NTSSPI, KERBEROS*.

For example, if the configuration file for a service includes the line:

```
SEC_WRITE_AUTHENTICATION = REQUIRED
```

The service must authenticate the client for any communciation that requires the *`WRITE`* context.

If the configuration file for a service includes the following line:

```
SEC_DEFAULT_AUTHENTICATION = REQUIRED
```

Provided it does not include any other authentication security configuration variables, this will define authentication over all contexts.

Where a specific policy is defined, the more specific value takes precedence over the default value.

Authentication and encryption can be accomplished using a variety of methods or protocols. Which protocol is selected is determined during the security negotiation process. To provide a list of available methods, use the following configuration parameters:

```
SEC_context_AUTHENTICATION_METHODS
SEC_context_CRYPTO_METHODS
```

These parameters will accept a comma- or space-delimited list of possible methods to use.

3.4. Authentication methods

The following methods of authentication are available:

- SSL (**`SSL`**)
- Kerberos (**`KERBEROS`**)
- Password (**`PASSWORD`**)
- Filesystem (**`FS`**)
- Remote filesystem (**`FS_REMOTE`**)
- Windows (**`NTSSPI`**)
- Claim-To-Be (**`CLAIMTOBE`**)
- Anonymous (**`ANONYMOUS`**)

SSL

SSL authentication is based on mutual X.509 certificates. When a connection is initiated, both the client and the server must verify the signature on the certificate.

The names and locations of keys and certificates are defined in the configuration files, using these variables:

- **`AUTH_SSL_CLIENT_CERTFILE`** specifies the location of the client (the process that initiates the connection) certificate file
- **`AUTH_SSL_SERVER_CERTFILE`** specifies the location for the server (the process that receives the connection) certificate file

- **`AUTH_SSL_CLIENT_KEYFILE`** specifies the location of the client key
- **`AUTH_SSL_SERVER_KEYFILE`** specifies the location of the server key
- **`AUTH_SSL_CLIENT_CAFILE`** specifies a path and file name for the location of client certificates issued by trusted certificate authorities
- **`AUTH_SSL_SERVER_CAFILE`** specifies a path and file name for the location of server certificates issued by trusted certificate authorities
- **`AUTH_SSL_CLIENT_CADIR`** specifies a directory containing client certificates that have been prepared using the OpenSSL **`c_rehash`** utility
- **`AUTH_SSL_SERVER_CADIR`** specifies a directory containing server certificatesthat have been prepared using the OpenSSL **`c_rehash`** utility

Kerberos

To use Kerberos for authentication, the Kerberos domain must be mapped to a Condor User ID (UID) domain. This is done by specifying the path to a Kerberos-specific map file in the configuration file, using the syntax:

```
KERBEROS_MAP_FILE = /path/to/etc/condor.kmap
```

Within the map file, the syntax is:

```
KERB.REALM = UID.domain.name
```

An example map file, containing two entries:

```
CS.WISC.EDU   = cs.wisc.edu
ENGR.WISC.EDU = ee.wisc.edu
```

If a map file has been defined, all permitted realms must be explicitly mapped. If a map file is not specified, then the Kerberos realm will be assumed to be the same as the Condor UID domain.

It is also possible to specify a unique name for assigning a set of credentials. This is done by specifying the value in the configurarion file, using the **`KERBEROS_SERVER_PRINCIPAL`** parameter. If a unique name is not defined, then it will default to *`host`*.

The name is used to define the server principal. Condor will use the defined name to calculate the server principal in the following way:

If the **`KERBEROS_SERVER_PRINCIPAL`** parameter is set as:

```
KERBEROS_SERVER_PRINCIPAL = condor-daemon
```

Condor will define the server principal as:

```
condor-daemon/the.host.name@YOUR.KERB.REALM
```

This example shows how to configure settings for Kerberos authentication. This will create a situation where all communications require authentication at the *`WRITE`* or *`ADMINISTRATOR`* level.

```
SEC_WRITE_AUTHENTICATION                 = REQUIRED
SEC_WRITE_AUTHENTICATION_METHODS         = KERBEROS
SEC_ADMINISTRATOR_AUTHENTICATION         = REQUIRED
SEC_ADMINISTRATOR_AUTHENTICATION_METHODS = KERBEROS
```

Example 3.2. Using Kerberos authentication

Note

Kerberos authentication requires root access to some files. Currently, the only supported way to use Kerberos authentication is to start Condor services as the root user.

Password

The password method provides authentication through a shared secret. This is useful where strong security is required, but an existing Kerberos or X.509 infrastructure does not exist. It is used only for authentication between services. The shared secret is referred to as the *pool password*.

To use password authentication, store the pool password on the local machine that is running the services. Define the location of the pool password in the configuration file using the **`SEC_PASSWORD_FILE`** parameter.

Generate a password file, using the command:

```
condor_store_cred -f /path/to/password/file
```

To store the pool password, use the **`-c`** option:

```
condor_store_cred -c add
```

This command will prompt for the password before storing it on the local machine. This makes it available for services to use it for authentication. This command will only work if the **`condor_master`** is running.

To store a pool password, the *`CONFIG`* security context is required. The pool password can be set remotely, but this method is only recommended if it takes place using an encrypted channel.

Filesystem

Filesystem authentication uses the ownership of a file to verify identity. A service that is attempting to authenticate a client will request that the client write a file to a **`/tmp`** directory. The service then checks the ownership of the file. If the file permissions match, the identity is verified and the file system becomes trusted.

Remote Filesystem

Similar to filesystem authentication, remote filesystem authentication uses the ownership of a file to verify the identity of a client. In this case, a service authenticating a client requires the client to write a file in a specific location. The location is not restricted to the **`/tmp`** directory, but is specified by the **`FS_REMOTE_DIR`** configuration variable.

This authentication method is only appropriate for clients and services that are on the same physical computer.

Windows

Systems running Microsoft Windows can use a proprietary authentication method that uses the SSPI interface to enforce the NT LAN Manager (NTLM). This authentication method is based on a "challenge and response" model, with the user password used as a key.

This authentication method is only appropriate for clients and services that are on the same physical computer. It should not be used for authentication between two computers.

Claim-To-Be

Claim-To-Be authentication accepts any identity claimed by the client. It is included for testing purposes only, and is not recommended for live systems.

Anonymous

Anonymous authentication causes authentication to be skipped entirely. It is included for testing purposes only, and is not recommended for live systems.

3.5. Encryption

Encryption provides privacy during communications. The client and the daemon can be configured to use encryption where required.

The client uses these parameters to enable or disable encryption:

- **`SEC_DEFAULT_ENCRYPTION`**
- **`SEC_CLIENT_ENCRYPTION`**

The service uses these parameters 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**

If a service has the following parameter set, any communication that will change the configuration of the service must be encrypted:

```
SEC_CONFIG_ENCRYPTION = REQUIRED
```

If a service has the following parameter set, and does not contain any other security configuration for encryption, all communication over all contexts must be encrypted:

```
SEC_DEFAULT_ENCRYPTION = REQUIRED
```

Where encryption has been requested, then a mutually agreeable method of encryption must be negotiated by the parties to the communication. A list of acceptable methods in a client can be defined using the following parameters:

- **SEC_DEFAULT_CRYPTO_METHODS**
- **SEC_CLIENT_CRYPTO_METHODS**

A list of acceptable methods in a service can be defined using the following parameters:

- **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**

Acceptable methods must be provided as a comma-separated list. The higher in the list a method is, the more preferred that method. Currently, possible values are:

- *3DES*

* *BLOWFISH*

3.6. Integrity

In order to check that the messages being sent and received have not been tampered with, it is possible to perform an integrity check. This will look for any additions or deletions within the message.

Through configuration macros, both the client and the daemon can specify whether an integrity check is required of further communication.

The client uses these parameters to enable or disable an integrity check:

* **SEC_DEFAULT_INTEGRITY**
* **SEC_CLIENT_INTEGRITY**

The service uses these parameters 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**

If a service has the following parameter set, any communication that will change the configuration of the service must have its integrity checked:

```
SEC_CONFIG_INTEGRITY = REQUIRED
```

If a service has the following parameter set, and does not contain any other security configuration for integrity checking, all communication over all contexts must have their integrity checked:

```
SEC_DEFAULT_INTEGRITY = REQUIRED
```

Curently, the only available method for checking integrity is by using an MD5 checksum. Use of this method is implied whenever an integrity check is requested.

Remote configuration

The remote configuration feature simplifies configuration and management of a pool. It allows a group of machines to be quickly and easily configured. The feature uses a central configuration store to maintain node configurations and provide notifications to managed nodes when necessary.

The remote configuration feature has three components:

Configuration store
: The *configuration store* is a central repository for configuration data. It maintains individual configuration entities and any configurations applied to the nodes it knows about.

Managed nodes
: The *nodes* managed by the configuration store. Each node in the store represents a physical machine to be managed. They check in with the store to retrieve configuration updates.

Tools
: There are a number of tools that can be used to interact with the configuration store. The tools are used to configure entities in the store which are then applied to the nodes.

The three components communicate using AMQP (advanced message queuing protocol). To facilitate this, an AMQP broker needs to be provided in a location that is available to all three components. Without this, remote configuration will not work correctly.

4.1. The Configuration Store

The configuration store is a central repository for configuration data. Before it can be used to send configurations out to nodes, it needs to be installed and initialized. Once a configuration has been stored, it can be put under version control, which enables configurations to be quickly and easily re-applied as necessary. The store also allows configurations to be modified without applying the changes to managed nodes. The store will send out notifications to the nodes that require it only once the configuration has been activated.

Installing and initializing the configuration store

1. Install the following packages with the **`yum`** command:

   - **`wallaby`**
   - **`wallaby-utils`**
   - **`condor-wallaby-base-db`**
   - **`condor-wallaby-tools`**

   ```
   # yum install wallaby wallaby-utils condor-wallaby-base-db condor-wallaby-tools
   ```

2. Start the wallaby service:

   ```
   # service wallaby start
   ```

3. Initialize the store with the default database for condor:

   ```
   $ wallaby load /var/lib/condor-wallaby-base-db/condor-base-db.snapshot
   ```

   The default database contains some common parameters and features with default values that can be used to build a condor pool.

   For more information about the metadata associated with various features used in the remote configuration feature, see *Appendix C, Feature Metadata*.

The configuration store records configurations as a series of entries in a database. Each entry contains certain elements, which can be paired with metadata. The elements are:

Parameters

A *parameter* is a key-value pair with associated metadata that is placed in condor's configuration file to alter how condor operates. A parameter can be dependent on other parameters, or have conflicts with other parameters.

Metadata for parameters

The metadata for parameters provides information about the parameter. It also provides useful information to the store when it sends notifications about configuration changes. Metadata for parameters can contain the following information:

`name`
: A unique string representing the name of the parameter.

`type`
: A one word string defining the type of parameter. Currently this is for documentation purposes only.

`default`
: The default value of the parameter. If no value is specified when the parameter is applied to a feature or node, this value can be used.

`description`
: A short text description of the parameter. This is for for documentation purposes only.

`conflicts`
: A list of other parameters that this parameter conflicts with.

`depends`
: A list of other parameters that this parameter depends upon.

`level`
: An integer representing the visibility level of the parameter.

`must_change`
: A boolean value. Defines if the parameter must be given a value when it is applied to a feature, group, or node in order to be valid. Parameters that must be given a value can not have a default value set.

`restart`
: A boolean value. Defines if the subsystems that rely on this parameter need to be restarted for the changes to be detected. If *`false`*, a **`condor_reconfig`** will be issued instead of a **`condor_restart`** for those subsystems.

Features

A *feature* is a group of parameters and their values. A parameter can have a specific value in a feature, or it can use the parameter's default value. Features can include, depend upon, or conflict with other features.

Metadata for features

Metadata for features can contain the following information:

`name`
: A unique string representing the name of the feature.

`params`
: Maps parameter names to the values that the feature contains. Values can be explicitly set, or left empty. If left empty, a default value will be obtained from the metadata of the parameter.

`conflicts`
: A list of other features that this feature conflicts with.

`depends`
: A list of other features that this feature depends upon.

`includes`
: A list of other features that this feature includes. The order of the list denotes the priority the configuration store will use to resolve common parameters with different values.

Nodes

A *node* represents a physical machine that will have configurations applied to it. Any feature or parameter applied to an individual node will take highest priority when determining configuration values for that node.

Metadata for nodes

The metadata information for nodes:

`name`
: A unique string representing the name of the node. The name must match the fully qualified hostname of the node it represents.

`memberships`
: A list of groups of which the node is a member.

Groups

A *group* is a group of nodes that will have parameters and features applied to it. Values for explicity-set group parameters will take priority over values for feature parameters. The configuration store has a built-in group: the **`Internal Default Group`**, which will always exist and be applied to all nodes within the store at the lowest priority.

Groups do not have associated metadata.

Subsystems

A *subsystem* is a program that is affected by parameters in the store. The subsystems determine which condor daemons will be acted upon when a configuration change is activated.

Metadata for subsystems

The metadata information for subsystems:

`name`
: A unique string representing the name of the subsystem.

`params`
: A list of parameters that rely on this subsystem. If a parameter in this list changes, the subsystem will be reconfigured or restarted upon a valid activation, depending upon the value set in the parameter's *`restart`* field.

Dependencies, conflicts, and includes

Features and parameters have metadata associated with them that can determine how they interact with other features and parameters:

Dependency
: A dependency is a condition that must be satisfied for the configuration to be valid. When an entity depends on another entity, the dependency must be set in the configuration separately in order for the configuration to be valid. Dependencies are applied in priority order, with lowest priority being applied first.

  Availability: Features, Parameters

Conflict
: A conflict can not exist anywhere in the configuration for the configuration to be valid.

  Availability: Features, Parameters

Include
: An include is similar to a dependency, except the condition is automatically resolved by the configuration store. If an entity includes another entity, the included entity does not need to be explicitly set in the configuration for the configuration to be valid. Includes are applied in priority order, with lowest priority being applied first.

  Availability: Features

Snapshot

A *snapshot* is a copy of the state of the store at a particular time. It contains all the parameters, features, nodes, groups, and subsystems, along with their respective metadata, as at the time the snapshot was taken. When the snapshot is loaded into the store, all the elements will be set as they are in the snapshot, and any changes or elements not contained in the snapshot will be lost.

Priorities

The configuration store use priorities to determine the order in which to inspect features and groups when determining a node's configuration. The **`Internal Default Group`** will always be the lowest priority group that a node is a member of, so it will be inspected first. The store will then evaluate the priority of the groups that the node is a member of, and finally evaluate any features or parameters applied to the node itself. If a parameter has multiple values set in multiple features or groups, the value given in the node's configuration will be the one determined by the highest priority group or feature.

Configuring the store using **`condor_configure_store`**

The **`condor_configure_store`** tool is used to add, remove, and edit parameters, features, groups, subsystems, and nodes in the configuration store. Only one action (add, remove, or edit) can be performed with each command, but multiple targets (parameters, features, groups, subsystems, or nodes) can be acted upon each time.

The **`condor_configure_store`** tool does not have to run on the same machine as the configuration store, nor the same machine as the broker the configuration store is communicating with. It will look for the AMQP broker on the machine it is running on by default, but it can be instructed to look for the broker in other locations, even if it is a non-standard port.

1. To add entities to the configuration store, use the **`condor_configure_store`** command with the **`--add`** or **`-a`** option, the target type, and the target:

   ```
   $ condor_configure_store -a -f feature1 -n node1,node2
   ```

   This example adds a feature called *`feature1`* and two nodes called *`node1`* and *`node2`* to the configuration store.

   Adding entities into the configuration store will invoke a text editor for entering and editing metadata about them. The text editor is defined in **`$EDITOR`**, and will default to vi. See *Editing metadata* for information about using the editor.

   After modifications have been saved, the tool will prompt for any additional instructions it requires. Follow the prompts to continue.

2. Once entities have been added, they can be edited using the **`--edit`** or **`-e`** option, the target type, and the target:

   ```
   $ condor_configure_store -e -s subsys1,subsys2 -p param1
   ```

   This example invokes a text editor to change the parameters of subsystems called *`subsys1`* and *`subsys2`*, and a parameter called *`param1`*.

   After modifications have been saved, the tool will prompt for any additional instructions it requires. Follow the prompts to continue.

Note

When adding or editing parameters for features, if a blank value (`""`) is given, the **`condor_configure_store`** tool will ask if the parameter should use the default value defined in the parameter's metadata.

3. To remove entities from the configuration store, use the **`--delete`** or **`-d`** option, the target type, and the target:

```
$ condor_configure_store -d -s subsys1,subsys2 -p param1 -f feature1 -n
node1,node2
```

   This example removes subsystems named *`subsys1`* and *`subsys2`*, a parameter named *`param1`*, a feature named *`feature1`*, and nodes named *`node1`* and *`node2`* from the configuration store.

4. To read the full help menu, use the **`--help`** or **`-h`** option:

```
$ condor_configure_store -h
```

Editing metadata

The metadata is edited using the YAML format. The three types of input are handled differently by YAML:

String

Any input that is not a list or a map is interpreted as a string. When a string contains a series of alphabetic characters (letters), it can optionally be surrounded by either single or double quotation marks. When the string is a series of numeric characters only, it must be surrounded by either single or double quotation marks.

An empty string is represented by quotation marks with no content between them (`''` or `""`).

List

A list is an ordered set of values. A list is comprised of one entry per line, with each line beginning with a hyphen (`-`) followed by a single whitespace and the value. For example:

```
a_list:
- value1
- value2
```

An empty list is represented by square brackets with no content between them (`[]`).

Map

A map is a set of name-value pairs. It is comprised of one name-value pair per line, with each pair seperated by a colon (`:`) and a single whitespace character. For example:

```
a_map:
 value1: This is a string
```

```
value2: '4'
```

An empty map is represented by curly braces with no content between them (`{}`)

4.2. Nodes

The **`condor-wallaby-client`** is installed on each node and is used to manage configurations for that node. It installs a configuration file in the MRG Grid local configuration directory, which enables it to control configuration for the node. The **`condor-wallaby-client`** package contains a service that will check in with the store and listen for configuration change notifications. When it receives a new configuration from the store, the service will write it into the local configuration file for the node. The location of the local configuration file is defined in the configuration file installed by the **`condor-wallaby-client`** package.

Managed nodes are considered either *provisioned* or *unprovisioned*. Nodes explicitly added to the store using remote tools are provisioned nodes. Nodes that have checked in with the store but have not explicitly been added are unprovisioned nodes. Unprovisioned nodes receive the configuration defined in the default group. A node must be explicitly added to the store with **`condor_configure_store`** (even if it already exists in the store) in order to change from unprovisioned to provisioned. Nodes are represented by their fully qualified domain names, and each node name in the store must be unique.

Installing and configuring the remote configuration client

1. To allow a machine to be managed with remote configuration, install the following package with the **`yum`** command:

   - **`condor-wallaby-client`**

   ```
   # yum install condor-wallaby-client
   ```

2. The remote configuration client needs to be told the location of the broker that is communicating with the configuration store. Open the condor local configuration directory and create or edit a configuration file with the following parameter, and the IP address or hostname of the machine running the broker:

   ```
   QMF_BROKER_HOST = wallaby_broker
   ```

3. Restart condor before attempting to configure nodes.

Applying configurations to a node with `condor_configure_pool`

The **`condor_configure_pool`** tool is used to apply entities in the configuration store to physical nodes. It is also used to manage configurations within the configuration store. Only one node or group can be acted upon with each command, but multiple features and parameters can by be acted upon each time.

The **`condor_configure_pool`** tool does not have to run on the same machine as the configuration store, nor the same machine as the broker the configuration store is communicating with. It will look for the AMQP broker on the machine it is running on by default, but it can be instructed to look for the broker in other locations, even if it is a non-standard port.

1. To display entities in the configuration store and their metadata, use the **`condor_configure_pool`** command with the **`--list-all-`*`type`*** option. It is possible to list more than one type by using successive commands:

   ```
   $ condor_configure_pool --list-all-nodes --list-all-subsystems
   ```

   This example will list all nodes and subsystems, and their metadata.

2. To apply entities to a node, use the **`-a`** option with the names of the entities and the target node:

   ```
   $ condor_configure_pool -a -n node1 -f feature1 -p param1,param2
   ```

   This example applies a feature called *`feature1`* and parameters called *`param1`* and *`param2`* to a node called *`noe1`*.

   Note

   To modify the configuration of the configuration store's Default Group, which affects all nodes the configuration store knows about, use the **`--default-group`** option as the target instead of a specific node or group name.

   When adding parameters to a node or group of nodes, the tool will prompt for the parameter values before asking to commit the changes to the configuration store. Answer *y* to instruct the tool to begin making the changes.

   The tool will prompt for any additional instructions it requires. Follow the prompts to continue.

   Once the changes have been applied, the tool will ask if the configuration should be saved. Answer *y* to confirm and provide a name for the configuration.

   The tool will ask if the changes should be activated. Answer *y* to instruct the tool to validate the changes and push them out to the pool.

3. A configuration change can be activated when it is made, or it can be activated at a later time. This allows multiple changes to be made, and then validated and applied all at once. To activate changes, use the **`--activate`** option:

   ```
   $ condor_configure_pool --activate
   ```

   The configuration store will validate the configuration and then push it out to all affected nodes. If the validation test does not pass, it will return an error message and exit.

   Note

   Changes that are activated using the **`--activate`** option will not generate a snapshot if a configuration is successfully activated.

4. To remove entities from the configuration store, use the **`--delete`** or **`-d`** option, the target type, and the target:

   ```
   $ condor_configure_pool -d -g group1 -f feature1,feature2
   ```

   This example removes features named *`feature1`* and *`feature2`* from a group of nodes called *`group1`*.

5. When changes are activated through the **`-a`** option, a snapshot is generated. Snapshots can also be taken at any time using the **`--take-snapshot`** option with an appropriate name:

   ```
   $ condor_configure_pool --take-snapshot "A snapshot name"
   ```

   This example will take a snapshot called *`A snapshot name`*.

   Once a snapshot exists, it can be loaded using the **`--load-snapshot`** option with the name of the snapshot to be loaded:

   ```
   $ condor_configure_pool --load-snapshot "A snapshot name"
   ```

   This example will load the snapshot called *`A snapshot name`*.

   Note
   Loading a snapshot will not instruct the configuration store to activate the configuration, or push any changes out to nodes. To apply the configuration changes in the snapshot, activate the configuration using the **`--activate`** option.

   Existing snapshots can be removed from the store using the **`--remove-snapshot`** option with the name of the snapshot to be removed:

   ```
   $ condor_configure_pool --remove-snapshot "A snapshot name"
   ```

   This example will remove the snapshot called *`A snapshot name`*.

6. To read the full help menu, use the **`--help`** or **`-h`** option:

   ```
   $ condor_configure_pool -h
   ```

4.3. Tools

There are several tools for use with the remote configuration tool. They do not need to be located on the same machine as the store, but they all must be provided by the AMQP broker that is connected to the store. If no broker information is provided, the tools will look for a broker on the machine they are run on.

`condor_configure_store`
: The **`condor_configure_store`** tool is used to configure parameters, features, groups, subsystems, and nodes in the store. Only one action can be performed at a time with this tool.

`condor_configure_pool`
: The **`condor_configure_pool`** tool is used to apply configurations to a specific node or group of nodes. It uses the parameters, features, groups, and nodes stored in the configuration store by the **`condor_configure_store`** tool. Only one node or group can be acted upon at a time, but multiple features and parameters can by be added at once.

Wallaby tools

The wallaby tool includes a set of useful commands for interacting with wallaby. All wallaby commands use the following syntax:

```
wallaby [broker options] command [command options]
```

`wallaby dump` ***`[outfile]`***
: The **`wallaby dump`** command is used to export objects in the store into plain text. The output can be placed into a file and loaded back into the store with **`wallaby load`**.

`wallaby load` ***`[file]`***
: The **`wallaby load`** command is used to load a file generated by **`wallaby dump`** into the configuration store. When a new database is loaded into the store, it will replace the entire contents of the store with the new information.

`wallaby inventory`
: The **`wallaby inventory`** tool is used to list details of the nodes being managed by the configuration store. It provides the node name, information about the last time the node checked in with the store, and whether the node was explicitly provisioned in wallaby or whether it checked in for a configuration before wallaby knew about it.

4.4. Remote configuration example

This example uses the remote configuration tool to configure a group of five nodes running condor. The configured pool contains a central manager, one scheduler, and three execute nodes. Any unprovisioned nodes are configured to run only the master daemon, but will report to the central manager.

Create the *`Workers`* group, and add all five nodes to the store:

```
$ condor_configure_store -a -g Workers -n node1,node2,node3,node4,node5
```

Add *`node3`*, *`node4`*, and *`node5`*, to the *`Workers`* group by setting their node membership in the editor:

```
memberships:
- Workers
```

Make *`node1`* the central manager. Answer *`N`* when asked to activate the changes:

```
$ condor_configure_pool -n node1 -a -f CentralManager
```

Make *node2* the scheduler. Answer *N* when asked to activate the changes:

```
$ condor_configure_pool -n node2 -a -f Scheduler
```

Make all nodes in the *Workers* group execute nodes. Answer *N* when asked to activate the changes:

```
$ condor_configure_pool -g Workers -a -f ExecuteNode
```

Make all nodes that check in with the store run only the master daemon:

```
$ condor_configure_pool --default-group -a -f Master
```

The tool will prompt for a value for **CONDOR_HOST**. This must be set for the configuration to be valid, and in this configuration should be *node1*. Answer *Y* when asked to set the parameter. If the configuration is correct, answer *Y* when asked to activate the changes.

In this example, the activation will fail for all nodes because they are missing a dependency. In order for the unprovisioned nodes to be able to check in with the central manager they will need the **NodeAccess** feature. This can be resolved by setting it on the default group:

```
$ condor_configure_pool --default-group -a -f NodeAccess
```

The tool will prompt for values for **ALLOW_READ** and **ALLOW_WRITE**. These must be set for the configuration to be valid. Answer *Y* when asked to set the parameters. If the configuration is correct, answer *Y* when asked to activate the changes.

The configuration is now activated and a named snapshot has been created. This can be verified by listing the snapshots:

```
$ condor_configure_pool --list-all-snapshots

Snapshots:
Automatically generated snapshot at date time -- hash
```

Example 4.1. Using the remote configuration tool

Jobs

A *job* is a piece of work to be accomplished. A job must include a *submit description file* which provides information to MRG Grid that helps control how the work is handled, including restrictions on the type of execute nodes are able to run the job. The work to be accomplished must be able to run without interactive user input.

5.1. Steps to submitting a job

5.1.1. Preparing the job

A job can be anything that will execute in the appropriate universe, such as a binary, Java class file, virtual machine, script, or other executable. Determine the input and output files required, and note their locations. Test the job, to ensure that it works as expected.

Standard input (STDIN) and console output (STDOUT and STDERR) can be redirected to and from files. Any files needed to perform STDIN functions must be created before the job can be submitted. They should also be tested to make sure they will run correctly.

5.1.2. Choosing a universe

MRG Grid uses an execution environment, called a *universe*. Jobs will run in the vanilla universe by default, unless a different universe is specified in the submit description file.

Currently, the following universes are supported:

- Vanilla
- Java
- VM (for Xen and KVM)
- Grid
- Scheduler
- Local
- Parallel

Vanilla universe

The vanilla universe is the default universe, and has very few restrictions.

If a vanilla universe job is partially completed when the remote machine has to be returned, or fails for some other reason, MRG Grid will perform one of two actions. It will either suspend the job, in case it can complete it on the same machine at a later time, or it will cancel the job and restart it again on another machine in the pool.

Java universe

The java universe allows users to run jobs written for the Java Virtual Machine (JVM). 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. MRG Grid will automatically locate details such as finding the JVM binary and setting the classpath.

VM universe

The VM universe allows for the running of Xen or KVM virtual machine instances. A VM universe job's lifecycle is tied to the virtual machine that is being run.

Grid universe

The Grid Universe provides jobs access to external schedulers. For example, jobs submitted to EC2 are routed through the Grid Universe.

Scheduler universe

The scheduler universe is primarily for use with the **`condor_dagman`** daemon. It allows users to submit lightweight jobs to be run immediately, alongside the **`condor_schedd`** daemon on the host machine. Scheduler universe jobs are not matched with a remote machine, and will never be pre-empted.

The scheduler universe, however, offers few features and limited policy support. The local universe is a better choice for most jobs which must run on the submitting machine, as it offers a richer set of job management features, and is more consistent with the other universes.

Local universe

The local universe allows a 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 is executed immediately, on the machine where the job is submitted. Jobs submitted in the local universe will never be pre-empted.

Parallel universe

The parallel universe is used to run jobs that require simultaneous startup on multiple execution nodes, such as Message Passing Interface (MPI) jobs.

5.1.3. Writing a submit description file

A job is submitted for execution using **`condor_submit`**, which requires a file called a *submit description file*. The submit description file contains the name of the executable, the initial working directory, and any command-line arguments.

The submit description file must inform **`condor_submit`** how to run the job, what input and output to use, and where any additional files are located.

The submit description file includes information about:

- Which executable to run
- The files to use for keyboard and screen data
- The platform required to run the program
- The universe to use. If you are unsure which universe to use, select the vanilla universe.

- Where to send notification emails
- How many times to run a program

The following examples are common submit description files, demonstrating the syntax to use when creating the file.

This example submits a job called *`physica`*.

Since no platform is specified in this description file, MRG Grid will default to run the job on a machine which has the same architecture and operating system as the machine from which it was submitted. The submit description file does not specify input, output, and error commands, this will cause MRG Grid to use **`/dev/null`** for all **`STDIN`**, **`STDOUT`** and **`STDERR`**. A log file, called ***`physica`*`.log`** will be created. When the job finishes, its exit conditions will be noted in the log file. It is recommended that you always have a log file.

```
Executable     = physica
Log            = physica.log
Queue
```

Example 5.1. Basic submit description file

This example 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 for each job are written to the individual directories. The job will be run in the vanilla universe.

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

Initialdir     = run_1
Queue

Initialdir     = run_2
Queue
```

Example 5.2. Using multiple directories in a submit description file

This example queues 150 runs of program *`chemistria`*.

This job must be run only on Linux workstations that have greater than 32 megabytes of physical memory. If machines with greater than 64 megabytes of physical memory are available, the job should be run on those machines as a preference. This submit description file also advises 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. In this case, **`STDIN`**, **`STDOUT`**, and **`STDERR`** will refer to **`in.0`**, **`out.0`** and **`err.0`** for the first run of the program, and **`in.1`**, **`out.1`** and **`err.1`** for the second run of the program. A log file will be written to ***`chemistria`*`.log`**.

```
Executable     = chemistria
```

```
Requirements   = Memory >= 32 && OpSys == "LINUX" && Arch =="X86_64"
Rank        = Memory >= 64
Image_Size     = 28 Meg

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

Queue 150
```

Example 5.3. Specifying execution requirements in a submit description file

This example submits a job to be run in the VM universe using KVM.

```
Universe=vm
Executable=testvm
Log=$(cluster).vm.log
VM_TYPE=kvm
VM_MEMORY=512
KVM_DISK=/var/lib/libvirt/images/RHEL5.img:vda:w
Queue
```

Example 5.4. Specifying the VM universe in a submit description file

5.1.4. Submitting the job

Submit the job with the **`condor_submit`** command. Full details of the **`condor_submit`** command can be found on the condor_submit manual page.

5.1.5. Monitoring job progress

Jobs can be monitored in a number of different ways. To check the status of a job using the command-line, use the **`condor_status`** command.

Jobs can also be queried using the **`condor_q`** command.

5.1.6. Finishing a job

When a job has been succesfully completed, MRG Grid will send an email to the address given in the submit description file. If no email address exists in the file, it will use the address in the configuration settings instead. The email will contain information about the job, including the time it took to complete, and the resources used.

If there is a log file recorded for the job, it will record an exit status code. A full list of the exit codes is in *Section B.4, “Shadow exit status codes”*.

If a job needs to be removed before it has been completed, this can be achieved by using the **`condor_rm`** command.

5.2. Time scheduling for job execution

MRG Grid allows jobs to begin execution at a later time. This feature can be accessed by adding a deferral time to the submit description file. Jobs running on a Unix platform can also be set to run periodically.

Deferring jobs

Job deferral allow the submitter to specify an exact date and time at which a job is to begin. MRG Grid attempts to match the job to an execution machine as normal, however, the job will wait until the specified time to begin execution. Submitters can also provide details for how to handle a job that misses it's specified execution time.

The *deferral time* is defined in the submit description file as a Unix Epoch timestamp. Unix Epoch timestamps are the number of seconds elapsed since midnight on January 1, 1970, Coordinated Universal Time.

After a job has been matched and the files transferred to a machine for execution, MRG Grid checks to see if the job has a deferral time. If it does, and the time for execution is still in the future, the job will wait. While it waits, *`JobStatus`* will indicate that the job is running.

If a job reports that the time for execution is in the past - that is, the job has failed to execute when it should have - then the job is evicted from the execution machine and put on hold in the queue. This could occur if the files were transferred too slowly, or because of a network outage. This can be avoided by specifying a *deferral window* within which the job can still begin. When a job arrives too late, the difference between the current time and the deferral time is calculated. If the difference is within the deferral window, the job will begin executing immediately.

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.

If a job is waiting to begin execution and a **`condor_hold`** command is issued, the job is removed from the execution machine and put on hold.

Limitations to the job deferral feature

There are some limitations to the job deferral feature:

- Job deferral will not work with scheduler universe jobs. If a deferral time is specified in a job submitted to the scheduler universe, a fatal error will occur.
- Job deferral times are based on the execution machine's system clock, not the submission machine's system clock.
- A job's *`JobStatus`* attribute will always show the job as *`running`* when job deferral is used. As of the 1.3 release of MRG Grid, there is no way to distinguish between a job that is executing and a job that has been deferred and is waiting to begin execution.

Example submit description files

The following examples show how to set job deferral times and deferral windows.

This example starts a job on January 1, 2008 at 09:00:00 GMT.

To calculate the date and time as Unix epoch time on a Unix-based machine, use the **`date`** program from the shell prompt with the following syntax:

```
$ date --date "MM/DD/YYYY HH:MM:SS" +%s
```

You could also use an online time converter, such as the *Epoch Converter*[1].

January 1, 2008 at 09:00:00 GMT converts to 1199178000 in Unix epoch time. The line you will need to add to the submit description file is:

```
deferral_time = 1199178000
```

Example 5.5. Setting the deferral time using Unix epoch time

This example starts a job one minute from the submit time.

This parameter uses a value in seconds to determine the start time:

```
deferral_time = (CurrentTime + 60)
```

Example 5.6. Setting the deferral time in seconds from submission time

This example sets a deferral window of 120 seconds, within which a job can begin execution

This parameter uses a value in seconds to determine the length of the deferral window:

```
deferral_window = 120
```

Example 5.7. Setting a deferral window in the submit description file

This example schedules a job to begin on January 1st, 2010 at 09:00:00 GMT, and sets a deferral prep time of 1 minute.

The deferral_prep_time attribute delays the job from being matched until the specified number of seconds before the job is to begin execution. This prevents the job from being assigned to a machine long before it is due to start and unnecessarily tying up resources.

```
deferral_time      = 1262336400
deferral_prep_time = 60
```

Example 5.8. Setting a deferral prep time in the submit description file

5.3. Using custom kill signals

Condor terminates jobs by sending them a signal. By default, vanilla universe jobs are killed with a **`SIGKILL`** command. This does not allow the job to perform a graceful shutdown, and is referred to as a *hard-kill*. To avoid this, it is possible for each job to define a custom kill signal. In this case, when the job is killed, the custom signal will be sent first. This allows the job to perform necessary functions for a graceful shutdown, such as writing out summary data. The starter will wait a period of time after initiating the job termination before determining that the starter is not responding and needs to be killed.

Defining a custom kill signal

1. In the job description file, specify the custom kill signal with the **`kill_sig`** parameter:

[1] http://www.epochconverter.com/

```
kill_sig = 3
```

Where *3* is **`SIGQUIT`**, or:

```
kill_sig = SIGQUIT
```

2. By default, the starter will wait the number of seconds defined in the **`killing_timeout`** configuration variable, less one second. It is also possible to set a timeout value in the job description file, using the **`kill_sig_timeout`** parameter. The starter will wait the shorter of the two values.

ClassAds

Job submission is simplified through the use of *ClassAds*. ClassAds are used to advertise the attributes of individual jobs and each slot on a machine. MRG Grid then uses the ClassAds to match jobs to slots.

Note

Slots are the logical equivalent of the physical cores on a machine. For example, a quad-core workstation would have four slots - with each slot being a dedicated allocation of memory (note however that hyperthreading will generally double the amount of slots available - a quad-core machine with hyperthreading would have eight slots).

ClassAds for slots advertise information such as:

- available RAM
- CPU type and speed
- virtual memory size
- current load average

Slots also advertise information about the conditions under which it is willing to run a job, and what type of job it would prefer. Additionally, machines can specify which jobs they would prefer to run. All this information is held by the ClassAd.

ClassAds for jobs advertise the type of machine they need to execute the job. For example, a job may require a minimum of 128MB of RAM, but would ideally like 512MB. This information is listed in the jobs ClassAd and slots that meet those requirements will be ranked for matching.

MRG Grid continuously reads all the ClassAds, ranking and matching jobs and slots. All requirements for both sets of ClassAds must be fulfilled before a match is made. ClassAds are generated automatically by the **`condor_submit`** daemon, but can also be manually constructed and edited.

This example uses the **`condor_status`** command to view ClassAds information from the machines available in the pool.

```
$ condor_status

Name        Arch      OpSys   State      Activity   LoadAv Mem   ActvtyTime

adriana.cs x86_64    LINUX   Claimed    Busy       1.000  64     0+01:10:00
alfred.cs. x86_64    LINUX   Claimed    Busy       1.000  64     0+00:40:00
amul.cs.wi x86_64    LINUX   Owner      Idle       1.000  128    0+06:20:04
anfrom.cs. x86_64    LINUX   Claimed    Busy       1.000  32     0+05:16:22
anthrax.cs x86_64    LINUX   Claimed    Busy       0.285  64     0+00:00:00
astro.cs.w x86_64    LINUX   Claimed    Busy       0.949  64     0+05:30:00
aura.cs.wi x86_64    LINUX   Owner      Idle       1.043  128    0+14:40:15
```

Example 6.1. Using **`condor_status`** to view ClassAds

The **`condor_status`** command has options that can be used to view the data in different ways. The most common options are:

`condor_status -available`
: Shows only those machines that are currently available to run jobs.

`condor_status -run`
: Shows only those machines that are currently running jobs.

`condor_status -l`
: Lists the ClassAds for all machines in the pool.

Note

Use **`$ man condor_status`** for a complete list of options.

Constraints and preferences

Jobs are matched to resources through the use of *constraints* and *preferences*.

Constraints and preferences for jobs are specified in the submit description file using **`requirements`** and **`rank`** expressions. For machines, this information is determined by the configuration.

The **`rank`** expression is used by a job to specify which requirements to use to rank potential machine matches.

This example uses the **`rank`** expression to specify preferences a job has for a machine.

A job ClassAd might contain the following expressions:

```
Requirements = Arch=="x86_64" && OpSys == "LINUX"
Rank         = TARGET.Memory + TARGET.Mips
```

In this case, the job requires an computer running a 64 bit Linux operating system. Among all such computers, the job prefers those with large physical memories and high MIPS (Millions of Instructions Per Second) ratings.

Example 6.2. Using the **`rank`** expression to set constraints and preferences for jobs

Any desired attribute can be specified for the **`rank`** expression. The **`condor_negotiator`** daemon will satisfy the required attributes first, then deliver the best resource available by matching the rank expression.

A machine may also specify constraints and preferences for the jobs that it will run.

This example using the machine configuration to set constraints and preferences a machine has for a job

A machine's configuration might contain the following:

```
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
```

This machine will always run a job submitted by members of the *`ResearchGroup`* but will never run jobs owned by users *`rival`* and *`riffraff`*. Jobs submitted by *`Friends`* are preferred to foreign jobs, and jobs submitted by the *`ResearchGroup`* are preferred to jobs submitted by *`Friends`*.

Example 6.3. Using machine configuration to set constraints and preferences

Querying ClassAd expressions

ClassAds can be queried from the shell prompt with the **`condor_status`** and **`condor_q`** tools. Some common examples are shown here:

Note

Use **`$ man condor_status`** and **`$ man condor_q`** for a complete list of options.

This example finds all computers that have mmore than 100MB of memory and their keyboard idle for longer than 20 minutes

```
$ condor_status -constraint 'KeyboardIdle > 20*60 && Memory > 100'

Name        Arch      OpSys   State      Activity   LoadAv Mem  ActvtyTime

amul.cs.wi x86_64    LINUX   Claimed    Busy       1.000  128   0+03:45:01
aura.cs.wi x86_64    LINUX   Claimed    Busy       1.000  128   0+00:15:01
balder.cs. x86_64    LINUX   Claimed    Busy       1.000  1024  0+01:05:00
beatrice.c x86_64    LINUX   Claimed    Busy       1.000  128   0+01:30:02
[output truncated]


      Machines  Owner Claimed Unclaimed Matched Preempting

x86_64/LINUX           3     0       3         0       0          0
x86_64/LINUX          21     0      21         0       0          0
x86_64/LINUX           3      0       3         0       0          0
x86_64/LINUX        1      0       0         1       0          0
x86_64/LINUX        1      0       1         0       0          0

Total           29     0    28         1       0          0
```

Example 6.4. Using the **`condor_status`** command with the **`-constraint`** option

This example uses a regular expression and a ClassAd function to list specific information.

A file called **ad** contains ClassAd information:

```
$ cat ad
MyType = "Generic"
FauxType = "DBMS"
Name = "random-test"
Machine = "f05.cs.wisc.edu"
MyAddress = "<128.105.149.105:34000>"
DaemonStartTime = 1153192799
UpdateSequenceNumber = 1
```

The **`condor_advertise`** daemon is used to insert the generic ClassAd information into the file:

```
$ condor_advertise UPDATE_AD_GENERIC ad
```

You can now use **`condor_status`** to constrain the search with a regular expression containing a ClassAd function:

```
$ condor_status -any -constraint 'FauxType=="DBMS" && regexp("random.*", Name, "i")'

MyType               TargetType           Name

Generic              None                 random-test
```

Job queues can also be queried in the same way.

Example 6.5. Using a regex and a ClassAd function to list information

6.1. Writing ClassAd expressions

The primary purpose of a ClassAd is to make matches, where the possible matches contain constraints. To achieve this, the ClassAd mechanism will continuously carry out expression evaluations, where two ClassAds test each other for a potential match. This is performed by the **`condor_negotiator`** daemon. This section examines the semantics of evaluating constraints.

A ClassAd contains a set of *attributes*, which are unique names associated with expressions.

```
MyType  = "Machine"
TargetType = "Job"
Machine  = "froth.cs.wisc.edu"
Arch  = "x86_64"
OpSys  = "LINUX"
Disk  = 35882
Memory  = 128
KeyboardIdle = 173
LoadAvg  = 0.1000
Requirements = TARGET.Owner=="smith" || LoadAvg<=0.3 && KeyboardIdle>15*60
```

Example 6.6. A typical ClassAd

ClassAd expressions are formed by *literals*, *attributes* and other sub-expressions combined with *operators* and *functions*. ClassAd expressions are not statically checked. For this reason, the expressions **`UNDEFINED`** and **`ERROR`** are used to identify expressions that contain names of attributes that have no associated value or that attempt to use values in a way that is inconsistent with their types.

Literals

Literals represent constant values in expressions. An expression that contains a literal will always evaluate to the value that the literal represents. The different types of literals are:

Integer
: One or more digits (0-9). Additionally, the keyword *TRUE* represents *`1`* and *FALSE* represents *`0`*

Real
: Two sequences of continuous digits separated by a **`.`** character

String
: Zero or more characters enclosed within **`"`** characters. A **`\`** character can be used as an escape character

Undefined
: The keyword **`UNDEFINED`** represents an attribute that has not been given a value.

Error
: The keyword **`ERROR`** represents an attribute with a value that is inconsistent with its type, or badly constructed.

Attributes

Every expression must have a name and a value, together the pair is referred to as an *attribute*. An attribute can be referred to in other expressions by its name.

Attribute names are sequences of letters, numbers and underscores. They can not start with a number. All characters in the name are significant, but they are not case sensitive.

A reference to an attribute must consist of the name of the attribute being refered to. References can also contain an optional *scope resolution prefix* of either **`MY.`** or **`TARGET.`**

The expression evaluation is carried out in the context of two ClassAds, creating a potential for ambiguities in the name space. The following rules define the semantics of attribute references made by *`ClassAd A`* which is being evaluated in relation to *`ClassAd B`*:

If the reference contains a scope resolution prefix:

- If the prefix is **`MY.`** the attribute will be looked up in *`ClassAd A`*. If the attribute exists in *`ClassAd A`*, the value of the reference becomes the value of the expression bound to the attribute name. If the attribute does not exist in *`ClassAd A`*, the value of the reference becomes **`UNDEFINED`**

- If the prefix is **`TARGET.`** the attribute is looked up in *`ClassAd B`*. If the attribute exists in *`ClassAd B`* the value of the reference becomes the value of the expression bound to the attribute name. If the attribute does not exist in *`ClassAd B`*, the value of the reference becomes **`UNDEFINED`**

If the reference does not contain a scope resolution prefix:

- If the attribute is defined in *`ClassAd A`* the value of the reference is the value of the expression bound to the attribute name in *`ClassAd A`*

- If the attribute is defined in *`ClassAd B`* the value of the reference is the value of the expression bound to the attribute name in *`ClassAd B`*

- If the attribute is defined in the ClassAd environment, the value from the environment is returned. This is a special environment, not the standard Unix environment. Currently, the only attribute of the environment is *`CurrentTime`*, which evaluates to the integer value returned by the **`system call time(2)`**

- If the attribute is not defined in any of the above locations, the value of the reference becomes **`UNDEFINED`**

If the reference refers to an expression that is itself in the process of being evaluated, it will cause a circular dependency. In thise case, the value of the reference becomes **`ERROR`**

Operators

The unary negation operator of **`-`** takes the highest precedence in a string. In order, operators take the following precedence:

1. **-** (unary negation)
2. ***** and **/**
3. **+** (addition) and **-** (subtraction)
4. **< <= >=** and **>**
5. **== != =?=** and **=!=**
6. **&&**
7. **||**

The different types of operators are:

Arithmetic operators
: The operators *** / +** and **-** operate arithmetically on integers and real literals

  Arithmetic is carried out in the same type as both operands. If one operand is an integer and the other real, the type will be promoted from integer to real

  Operators are strict with respect to both **UNDEFINED** and **ERROR**

  If one or both of the operands are not numerical, the value of the operation is **ERROR**

Comparison operators
: The comparison operators **== != <= < >=** and **>** operate on integers, reals and strings

  The operators **=?=** and **=!=** behave similarly to **==** and **!=**, but are not strict. Semantically, **=?=** tests if its operands have the same type and the same value. For example, **10 == UNDEFINED** and **UNDEFINED == UNDEFINED** both evaluate to **UNDEFINED**, but **10 =?= UNDEFINED** will evaluate to **FALSE** and **UNDEFINED =?= UNDEFINED** will evaluate to **TRUE**. The **=!=** operator tests for not identical conditions

  String comparisons are case insensitive for most operators. The only exceptions are the operators **=?=** and **=!=** which perform case sensitive comparisons when both sides are strings

  Comparisons are carried out in the same type as both operands. If one operand is an integer and the other real, the type will be promoted from integer to real

  Strings can not be converted to any other type, so comparing a string and an integer or a string and a real results in **ERROR**

  The operators **== != <= <** and **>= >** are strict with respect to both **UNDEFINED** and **ERROR**

Logical operators
: The logical operators **&&** and **||** operate on integers and reals. The zero value of these types are considered **FALSE** and non-zero values **TRUE**

  Logical operators are not strict, and exploit the "don't care" properties of the operators to eliminate **UNDEFINED** and **ERROR** values when possible. For example, **UNDEFINED && FALSE** evaluates to **FALSE**, but **UNDEFINED || FALSE** evaluates to **UNDEFINED**

  Any string operand is equivalent to an **ERROR** operand for a logical operator. For example **TRUE && "string"** evaluates to **ERROR**

Pre-defined functions

ClassAd expressions can use predefined functions. Function names are not case sensitive. Function calls can also be nested or recursive.

This is a complete list of predefined functions. The format of each function is:

```
ReturnType FunctionName(ParameterType1 parameter1, ParameterType2 parameter2, ...)
```

The possible types are as listed in *Literals*. If the function can be any of these literal types, it is described as **AnyType**. Where the type is **Integer**, but only returns the value *1* or *0* (*True* or *False*), it is described as **Boolean**. Optional parameters are given in square brackets.

AnyType ifThenElse(**AnyType** *IfExpr*,**AnyType** *ThenExpr*, **AnyType** *ElseExpr*)

A conditional expression.

When *IfExpr* evaluates to **true**, return the value as given by *ThenExpr*

When **false**, return the value as given by *ElseExpr*

When **UNDEFINED**, return the value **UNDEFINED**

When **ERROR**, return the value **ERROR**

When *IfExpr* evaluates to **0.0**, return the value as given by *ElseExpr*

When *IfExpr* evaluates to a non-**0.0** or Real value, return the value as given by *ThenExpr*

When *IfExpr* evaluates to give a value of type **String**, return the value **ERROR**

Expressions are only evaluated as defined

If a number of arguments other than three are given, the function will return **ERROR**

Boolean isUndefined(**AnyType** *Expr*)

Returns *True* if *Expr* evaluates to **UNDEFINED**. Otherwise, returns *False*

If a number of arguments other than one is given, the function will return **ERROR**

Boolean isError(**AnyType** *Expr*)

Returns *True*, if *Expr* evaluates to **ERROR**. Otherwise, returns *False*

If a number of arguments other than one is given, the function will return **ERROR**

Boolean isString(**AnyType** *Expr*)

Returns *True* if *Expr* gives a value of type **String**. Otherwise, returns *False*

If a number of arguments other than one is given, the function will return **ERROR**

Boolean isInteger(**AnyType** *Expr*)

Returns *True*, if *Expr* gives a value of type **Integer**. Otherwise, returns *False*

If a number of arguments other than one is given, the function will return **ERROR**

Boolean isReal(**AnyType** *Expr*)

Returns *True* if *Expr* gives a value of type **Real**. Otherwise, returns *False*

If a number of arguments other than one is given, the function will return **ERROR**

`Boolean` isBoolean(**AnyType** *`Expr`*)

Returns *`True`*, if *`Expr`* returns an integer value of *`1`* or *`0`*. Otherwise, returns *`False`*

If a number of arguments other than one is given, the function will return **`ERROR`**

`Integer` int(**AnyType** *`Expr`*)

Returns the integer value as defined by *`Expr`*

Where the type of the evaluated *`Expr`* is **`Real`** the value is rounded down 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. If the result is not an integer, **`ERROR`** is returned

Where the evaluated *`Expr`* is **`ERROR`** or **`UNDEFINED`**, **`ERROR`** is returned

If a number of arguments other than one is given, the function will return **`ERROR`**

`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. If the result is not **`real`** **`ERROR`** is returned

Where the evaluated *`Expr`* is **`ERROR`** or **`UNDEFINED`**, **`ERROR`** is returned

If a number of arguments other than one is given, the function will return **`ERROR`**

`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 in a similar way 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 the format is not specified, it defaults to *%c* (local date and time representation).

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 use 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` string(**AnyType** *`Expr`*)

Returns the string that results from the evaluation of *`Expr`*

A non-**`string`** value will be converted to a **`string`**

Where the evaluated *`Expr`* is **`ERROR`** or **`UNDEFINED`**, **`ERROR`** is returned

If a number of arguments other than one is given, the function will return **`ERROR`**

`Integer` floor(**AnyType** *`Expr`*)

When the type of the evaluated *`Expr`* is **`Integer`**, returns the integer that results from the evaluation of *`Expr`*

When the type of the evaluated *`Expr`* is anything other than **`Integer`**, function **`real(Expr)`** is called. Its return value is then used to return the largest integer that is not higher than the returned value

Where the **`Real(Expr)`** returns **`ERROR`** or **`UNDEFINED`**, **`ERROR`** is returned

If a number of arguments other than one is given, the function will return **`ERROR`**

`Integer` ceiling(**AnyType** *`Expr`*)

When the type of the evaluated *`Expr`* is **`Integer`**, returns the integer that results from the evaluation of *`Expr`*

When the type of the evaluated *`Expr`* is anything other than **`Integer`**, function **`real(Expr)`** is called. Its return value is then used to return the smallest integer that is not less than the returned value

Where the **`Real(Expr)`** returns **`ERROR`** or **`UNDEFINED`**, **`ERROR`** is returned

If a number of arguments other than one is given, the function will return **`ERROR`**

`Integer` round(**AnyType** *`Expr`*)

When the type of the evaluated *`Expr`* is **`Integer`**, returns the integer that results from the evaluation of *`Expr`*

When the type of the evaluated *`Expr`* is anything other than **`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 values. In this case, the even valued integer is returned

Where the **`Real(Expr)`** returns **`ERROR`** or **`UNDEFINED`**, or the integer does not fit into 32 bits **`ERROR`** is returned

If a number of arguments other than one is given, the function will return **`ERROR`**

`Integer` random([**AnyType** *`Expr`*])

When the type of the optional argument *`Expr`* evaluates to **`Integer`** or **`Real`**, 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)`*

In all other cases, the function will return **`ERROR`**

If a number of arguments other than one is given, the function will return **`ERROR`**

`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)`**

If any argument evaluates to **`ERROR`** or **`UNDEFINED`**, **`ERROR`** is returned

`String` substr(**`String s, Integer`** *`offset`* [, **`Integer`** *`length`*])

Returns the substring **`s`**, from the position indicated by *`offset`*, with optional *`length`* characters

The first character within **`s`** is at offset *`0`*. If the *`length`* argument is not used, the substring extends to the end of the string

If *`offset`* is negative, the value of **`length - offset`** is used for *`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

If a number of arguments other than either two or three is given, the function will return **`ERROR`**

`Integer` strcmp(**AnyType** *`Expr1`*, **AnyType** *`Expr2`*)

Both arguments are converted to type **`String`** by **`string(Expr)`**

The return value is an integer that will be less than *`0`* if *`Expr1`* is less than *`Expr2`*

The return value will be equal to *`0`* if *`Expr1`* is equal to *`Expr2`*

The return value will be greater than *`0`* if *`Expr1`* is greater than *`Expr2`*

Case is significant in the comparison. Where either argument evaluates to **`ERROR`** or **`UNDEFINED`**, **`ERROR`** is returned

If a number of arguments other than two is given, the function will return **`ERROR`**

`Integer` stricmp(**AnyType** *`Expr1`*, **AnyType** *`Expr2`*)

This function is the same as the **`strcmp`** function, except that letter case is not significant

`String` toUpper(**AnyType** *`Expr`*)

The argument is converted to type **`String`** by the **`string(Expr)`**

The return value is a string, with all lower case letters converted to upper case

If the argument evaluates to **`ERROR`** or **`UNDEFINED`**, **`ERROR`** is returned

If a number of arguments other than one is given, the function will return **`ERROR`**

`String` toLower(**`AnyType`** *`Expr`*)
The argument is converted to type **`String`** by the **`string(Expr)`**

The return value is a string, with all upper case letters converted to lower case

If the argument evaluates to **`ERROR`** or **`UNDEFINED`**, **`ERROR`** is returned

If a number of arguments other than one is given, the function will return **`ERROR`**

`Integer` size(**`AnyType`** *`Expr`*)
Returns the number of characters in the string, after calling the **`string(Expr)`** function

If the argument evaluates to **`ERROR`** or **`UNDEFINED`**, **`ERROR`** is returned

If a number of arguments other than one is given, the function will return **`ERROR`**

The following functions contain string lists. *String delimiters* are used to define how the string list should be read. The characters in the string delimiter define the characters used to separate the elements within the string list.

This example uses the **`stringListSize`** function to demonstrate how a string delimiter of "`, |`" (a comma, followed by a space character, followed by a pipe) operates.

The function is given as follows:

```
StringListSize("1,2 3|4&5", ", |")
```

Firstly, the string is broken up according to the first delimiter - the comma character - resulting in the following two elements:

```
"1" and "2 3|4&5"
```

Now perform the same process, using the second delimiter - the space character - resulting in three elements:

```
"1", "2" and "3|4&5"
```

Finally, apply the third delimiter - the pipe character - resulting in a total of four elements:

```
"1", "2", "3" and "4&5"
```

Note that because the & character is not defined as a delimiter, the final group ("4&5") is considered only one element

Example 6.7. Using a string delimiter of "`, |`" with string lists

Note

The *`string delimiter`* is optional in the following functions. If no *`string delimiter`* is defined, the default string delimiter of "` ,`" (a space character, followed by a comma) is used.

`Integer` stringListSize(**`String`** *`list`* [, **`String`** *`delimiter`*])

Returns the number of elements in the **`String`** *`list`*, as delimited by the **`String`** *`delimiter`*

If one or both of the arguments is not a string, returns **`ERROR`**

If a number of arguments other than one is given, the function will return **`ERROR`**

`Integer` stringListSum(**`String`** *`list`* [, **`String`** *`delimiter`*]) *OR* **`Real`** stringListSum(**`String`** *`list`* [, **`String`** *`delimiter`*])

Returns the sum of all items in the **`String`** *`list`*, as delimited by the **`String`** *`delimiter`*

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, the return value is real

If any item is not either an integer or real value, the return value is **`ERROR`**

`Real` stringListAve(**`String`** *`list`* [, **`String`** *`delimiter`*])

Sums and returns the real-valued average of all items in the **`String`** *`list`*, as delimited by the **`String`** *`delimiter`*

If any item is neither an integer nor a real value, the return value is **`ERROR`**

A list containing no items returns the value *`0.0`*

`Integer` stringListMin(**`String`** *`list`* [, **`String`** *`delimiter`*]) *OR* **`Real`** stringListMin(**`String`** *`list`* [, **`String`** *`delimiter`*])

Returns the minimum value from all items in the **`String`** *`list`*, as delimited by the **`String`** *`delimiter`*

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, the return value is a real

If any item is neither an integer nor a real value, the return value is **`ERROR`**

A list containing no items returns the value *`UNDEFINED`*

`Integer` stringListMax(**`String`** *`list`* [, **`String`** *`delimiter`*]) *OR* **`Real`** stringListMax(**`String`** *`list`* [, **`String`** *`delimiter`*])

Returns the maximum value from all items in the **`String`** *`list`*, as delimited by the **`String`** *`delimiter`*

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, the return value is a real

If any item is neither an integer nor a real value, the return value is **`ERROR`**

A list containing no items 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 **`String`** *`delimiter`*

Returns *`FALSE`* if item *x* is not in the **`string`** *`list`*

Comparison is performed with the **`strcmp()`** function

If the arguments are not strings, the return value is **`ERROR`**

`Boolean` stringListIMember(**`String`** *x*, **`String`** *`list`* [, **`String`** *`delimiter`*])

This function is the same as the **`stringListMember`** function, except that the comparison is done with the **`stricmp()`** function, so letter case is not significant

The following functions contain a regular expression (*regex*) and an options argument. The options argument is a string of special characters that modify the use of the regex. The only accepted options are:

Option	Description
`I` or *`i`*	Ignore letter case
`M` or *`m`*	Modifies the interpretation of the carat (`^`) and dollar sign (`$`) characters, so that `^` matches the start of a string, as well as after each new line character and `$` matches before a new line character
`S` or *`s`*	Modifies the interpretation of the period (`.`) character to match any character, including the new line character.
`X` or *`x`*	Ignore white space and comments within the pattern. A comment is defined by starting with the `#` character, and continuing until the new line character.

Table 6.1. Options for use in functions

Note

For a complete list of regular expressions visit the *PCRE Library*[1]

`Boolean` regexp(**`String`** *`pattern`*, **`String`** *`target`* [, **`String`** options])

Returns *TRUE* if the **`String`** *`target`* is a regular expression as described by *`pattern`*. Otherwise returns *FALSE*

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

If any argument is not a **`String`** returns **`ERROR`**

`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`*. Otherwise returns *FALSE*

If any argument is not a **`String`**, or if *`pattern`* does not describe a valid regular expression, returns **`ERROR`**

To include the optional fourth argument options, a third argument of **`String`** *`delimiter`* is required. If a specific *`delimiter`* is not specified, the default value of " `,`" (a space character followed by a comma) will be used

`Integer` time()
: Returns the current Unix epoch time, which is equal to the ClassAd attribute **`CurrentTime`**. This is the time, in seconds, since midnight on January 1, 1970

`String` interval(**`Integer`** *`seconds`*)
: Uses seconds to return a string in the form of *`days+hh:mm:ss`* representing an interval of time. Leading values of zero are omitted from the string

6.2. Resource restriction

The **`condor_startd`** daemon is able to divide system resources amongst all available slots, by changing how they advertised to the collector for match-making purposes. This parameter will cause all jobs to execute inside a wrapper that will enforce limits on RAM, disk space, and swap space.

Important

This parameter prevents a job from using more resources than have been allocated to the slot running it. It is important to accurately specify the resources needed when submitting the job, to prevent errors in execution.

This example demonstrates resource restriction.

In this example, an execute node exists with the following resources:

- 2GB of RAM
- 100GB of disk space
- 100MB of swap space

This execute node has 2 slots with the resources evenly divided between them. This means that each slot advertises to the collector with the following resources:

- 1GB of RAM
- 50GB of disk space
- 50MB of swap space

Any job that requests more resources than what is available by one of these slots will not be matched. However, if a job is advertised with the following requirements, it will be matched:

- 1GB of RAM
- 50GB of disk space
- 50MB of swap space

If this job was matched and run, but eventually needed more than 1GB of RAM, 50GB of disk, or 50MB of swap space, nothing would prevent the job from consuming the resources it needs to run.

This can become a problem, as the other slot is still advertising half the system resources, even though less than half is now available.

If a second job is matched to the other available slot and attempts to use the resources it has requested, it is likely to create resource contention with the first job.

Setting and using resource restriction creates a wrapper for the job to run within. This means that the example job would not be able to consume more than 1GB of RAM, 50GB of disk space, or 50GB of swap space. If it did attempt to, job execution would fail.

Example 6.8. Resource restriction

Warning

Using this feature can result in hard failures in the application. Memory allocations or disk writes can fail if they attempt to use more resources than is allocated on the slot. Any jobs to be run with strict resource enforcement should be written in such a way that they are able to gracefully handle failures when requesting resources.

1. Create a file in the local configuration directory, and add the following lines:

   ```
   USER_JOB_WRAPPER=$(LIBEXEC)/condor_limits_wrapper.sh
   ```

2. Save the changes, and restart the condor service:

   ```
   # service condor restart
   Stopping condor services:                    [  OK  ]
   Starting condor services:                    [  OK  ]
   ```

Tracking Processes

MRG Grid needs to keep track of all processes created by every job. This is neccesary in order to provide resource usage statistics, and also so that processes can be properly cleaned up once a job has been completed. This is achieved by tracking a combination of parent/child process relationships, groups, and using the information in each job environment.

Tracking processes reliably can be difficult. When condor cleans up after executing a 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 attempts to track the CPU usage of all processes created by the job. There are a variety of mechanisms used to detect all processes, but the only way to catch them all is to run the job under a dedicated execution account. Otherwise, some processes will always be left behind. Sometimes, these processes can then create problems for the next job to be processed. Running jobs with a dedicated Condor user account or group can help increase the reliability of process tracking.

Setting up a dedicated user account

When a job is submitted, it is usually run under the user account that submitted it. However, there are some conditions which will make it run under the user **nobody**. The **nobody** user is often used by the system for other jobs, or non-condor tasks as well, which can make killing processes owned by **nobody** complicated. To avoid this issue, create a dedicated low-privilege user account for each job execution slot on every machine. These user accounts can then be used for running jobs instead of the **nobody** account.

1. Create the users: one for each job execution slot on each machine. In this example, the machine has two slots, so two users have been created: **`condorusr1`**, and **`condorusr2`**:

   ```
   # adduser condorusr1
   # adduser condorusr2
   ```

2. Create a file in the local configuration directory, and add the following lines:

   ```
   SLOT1_USER = condorusr1
   SLOT2_USER = condorusr2
   ```

3. Mark these users as dedicated by adding the *`DEDICATED_EXECUTE_ACCOUNT_REGEXP`* configuration variable to the file. This allows condor to kill all the processes belonging to these users when a job has been completed. The *`DEDICATED_EXECUTE_ACCOUNT_REGEXP`* configuration variable uses a regular expression to match the user accounts:

   ```
   DEDICATED_EXECUTE_ACCOUNT_REGEXP = condorusr[0-9]+
   ```

4. Finally, adjust the **`STARTER_ALLOW_RUNAS_OWNER`** configuration variable, so that it no longer runs jobs as the job owner, but uses the dedicated user accounts instead:

   ```
   STARTER_ALLOW_RUNAS_OWNER = False
   ```

5. To check if the dedicated user is being used as expected, check the log file for this entry:

```
Tracking process family by login "condorusr1"
```

Setting up a dedicated group

To be able to track processes regardless of the user account of the job they have submitted under, it is also possible to create a range of dedicated group IDs (GIDs). In this case, the dedicated GID is specified in a supplementary group list of a job's initial process. Because doing this requires root privileges, the job will not be able to create processes that condor cannot see.

1. Create a dedicated GID range, ensuring that the number of IDs available in the range matches the number of execution slots available on the machine. In this example, the machine has eight slots, so a range of eight GIDs has been created, from *750* to *757*:

   ```
   # groupadd -g 750-757
   ```

2. Create a file in the local configuration directory, and add the following lines:

   ```
   USE_GID_PROCESS_TRACKING = True
   MIN_TRACKING_GID = 750
   MAX_TRACKING_GID = 757
   ```

   Note

   Process tracking using GID requires use of the **condor_procd** daemon. If the *USE_GID_PROCESS_TRACKING* configuration variable is set to *True*, **condor_procd** will be used regardless of the setting for *USE_PROCD*.

Job Hooks

A *hook* is an external program or script invoked during the life cycle of a job. External programs or scripts allow MRG Grid to interface with external systems, and fetch work or perform other tasks that could not be accomplished with MRG Grid alone. This can result in an easier and more direct method of interfacing with an external scheduling system.

Hooks can also be useful where a job needs to be performed behind a firewall, but requires data from outside. The hook only needs an outbound network connection to complete the task, thereby being able to operate from behind the firewall, without the intervention of a connection broker.

Hooks can also be used to manage the execution of a job. They can be used to fetch execution environment variables, update information about the job as it runs, notify when it exits, or take special action if the job is evicted.

MRG Grid can be configured to look for jobs by using job hooks. Any job retrieved by the job hooks is evaluated to decide if it should be executed, and whether or not it should pre-empt any currently running jobs. If the resources are not available to run the retrieved job, it will be refused. Jobs retrieved by job hooks will not appear in any of the MRG Grid queue mechanisms (like **condor_q**).

When a job is accepted the **condor_startd** daemon will spawn a **condor_starter** daemon to manage the execution of the job. The job will then be treated as any other, and can potentially be pre-empted by a higher ranking job.

Job hooks related to fetching or evicting jobs are handled either by the **condor_startd**. Job hooks invoked during a job's lifecycle are handled by the **condor_starter** daemon.

The different types of hooks are:

HOOK_FETCH_WORK
: This hook returns any work to be considered by the **condor_startd** daemon. The *FetchWorkDelay* configuration variable determines how long the daemon will wait between attempts to fetch work.

HOOK_REPLY_FETCH
: When a job is retrieved with the **HOOK_FETCH_WORK** job hook, the **condor_startd** decides whether to accept or reject the fetched job and uses **HOOK_REPLY_FETCH** job hook to send notification of the decision.

  Importantly, this hook is advisory in nature. **condor_startd** will not wait for the results of **HOOK_REPLY_FETCH** before performing other actions. The output and exit status of this hook is ignored.

HOOK_EVICT_CLAIM
: **HOOK_EVICT_CLAIM** is invoked by **condor_startd** in order to evict a fetched job. This hook is also advisory in nature.

HOOK_PREPARE_JOB
: When a job is going to be run, **condor_starter** invokes **HOOK_PREPARE_JOB**. This job hook allows commands to be executed to set up the job environment, such as transferring input files.

  condor_starter will wait for **HOOK_PREPARE_JOB** to be returned before it attempts to execute the job. An exit status of *0* indicates that the job has been prepared succesfully. If the hook returns with an exit status that is not *0*, an error has occured and the job will be aborted.

`HOOK_UPDATE_JOB_INFO`

This hook is invoked periodically during the life of a job to update job status information. The period before the hook is invoked for the first time can be adjusted by changing the **`STARTER_INITIAL_UPDATE_INTERVAL`** configuration variable. After the initial interval, further intervals can be adjusted with the **`STARTER_UPDATE_INTERVAL`** configuration variable. Using the default values, the hook would be invoked for the first time eight seconds after the job has begun executing, and then every five minutes (600 seconds) thereafter.

`HOOK_JOB_EXIT`

This hook is invoked whenever a job exits - either through completion or eviction.

The **`condor_starter`** will wait for this hook to return before taking any further action.

When a hook is invoked, it will have certain privileges. Job hooks invoked from the **`condor_startd`** will have the same privileges as the condor user (or the privileges of the user running the **`startd`**, if that is a user other than Condor). Job hooks invoked by the **`condor_starter`** will have the same privileges as the job owner.

Defining the FetchWorkDelay Expression

The **`condor_startd`** daemon will attempt to fetch new work in two circumstances:

1. When **`condor_startd`** evaluates its own state; and
2. When the **`condor_starter`** exits after completing fetched work.

It is possible that, even if a slot is already running another job, it could be pre-empted by a new job, which could result in a problem known as *thrashing*. In this situation, every job gets pre-empted and no job has a chance to finish. By adjusting the frequency that **`condor_startd`** checks for new work, this can be prevented. This can be achieved by defining the **`FetchWorkDelay`** configuration variable.

The **`FetchWorkDelay`** variable is expressed as the number of seconds to wait in between the last fetch attempt completing and attempting to fetch another job.

This example instructs **`condor_startd`** to wait for 300 seconds (5 minutes) between attempts to fetch jobs, unless the slot is marked as *`Claimed/Idle`*. In this case, **`condor_startd`** should attempt to fetch a job immediately:

```
FetchWorkDelay = ifThenElse(State == "Claimed" && Activity == "Idle", 0, 300)
```

If the **`FetchWorkDelay`** variable is not defined, **`condor_startd`** will default to a 300 second (5 minute) delay between all attempts to fetch work, regardless of the state of the slot.

Example 8.1. Setting the **`FetchWorkDelay`** configuration variable

Using keywords to define hooks in configuration files

Hooks are defined in the configuration files by prefixing the name of the hook with a keyword. This allows a machine to have multiple sets of hooks, with each set identified by a keyword.

Each slot on a machine can define a separate keyword for the set of hooks that should be used. If a slot-specific keyword is not used, **`condor_startd`** will use the global keyboard defined in the **`STARTD_JOB_HOOK_KEYWORD`** configuration variable.

Note

Slots are the logical equivalent of the physical cores on a machine. For example, a quad-core workstation would have four slots - with each slot being a dedicated allocation of memory (note however that hyperthreading will generally double the amount of slots available - a quad-core machine with hyperthreading would have eight slots).

Once a job has been retrieved using the **`HOOK_FETCH_WORK`** job hook, the **`condor_startd`** daemon will use the keyword for that job to select the hooks required to execute it.

This is an example configuration file that defines hooks on a machine with four slots.

Three of the slots (slots 1-3) use the global keyword for running work from a database-driven system. These slots need to fetch work and provide a reply to the database system for each attempted claim.

The fourth slot (slot 4) uses a custom keyword to handle work fetched from a web service. It needs only to fetch work.

```
STARTD_JOB_HOOK_KEYWORD = DATABASE

SLOT4_JOB_HOOK_KEYWORD = WEB

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

WEB_HOOK_DIR = /usr/local/condor/fetch/web
WEB_HOOK_FETCH_WORK = $(WEB_HOOK_DIR)/fetch_work.php
```

Note that the keywords *DATABASE* and *WEB* are very generic terms. It is advised that you choose more specific keywords for your own installation.

Example 8.2. Using keywords when defining hooks

Policy Configuration

Machines in a pool can be configured through the **`condor_startd`** daemon to implement policies that perform actions such as:

- Start a remote job
- Suspend a job
- Resume a job
- Vacate a job
- Kill a job

Policy configuration is the at the heart of the balancing act between the needs and wishes of machine owners and job submitters. This section will outline how to adjust the policy configuration for machines in your pool.

Note

If you are configuring the policy for a machine with multiple cores, and therefore multiple slots, each slot will have its own individual policy expressions. In this case, the word *machine* refers to a single slot, not to the machine as a whole.

This chapter assumes you know and understand ClassAd expressions. Ensure that you have read *Chapter 6, ClassAds* before you begin.

9.1. Machine states and transitioning

Every machine is assigned a *state*, which changes as machines become available to run jobs. The six possible states are:

Owner
: The machine is not available to run jobs. This state normally occurs when the machine is being used by the owner. Additionally, machines begin in this state when they are first turned on

Unclaimed
: The machine is available to run jobs, but is not currently doing so

Matched
: The machine has been matched to a job by the negotiator, but the job has not claimed the machine

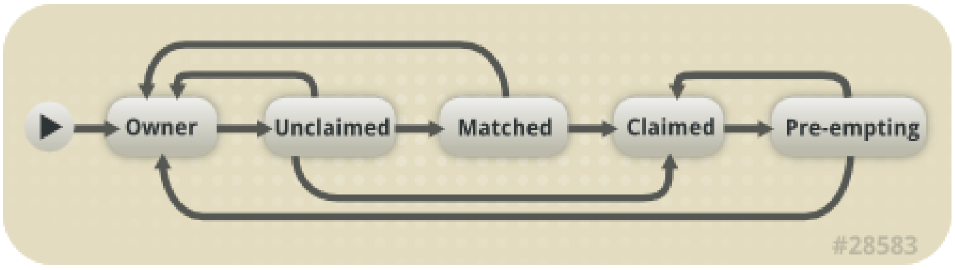
Claimed
: The machine has been claimed by a job. The job may be currently executing, or waiting to begin execution

Preempting
: The machine was claimed, but is now being pre-empted. This state is used to evict a running job from a machine, so that a new job can be started. This can happen for one of the following reasons:

  - The owner has started using the machine

- Jobs with a higher priority are waiting to run
- Another request that this resource would rather serve was found

The following diagram demonstrates the machine states and the possible transitions between them.



Possible transitions between machine states

`Owner` to `Unclaimed`
: This transition occurs when the machine becomes available to run a job. This occurs when the **`START`** expression evaluates to *`TRUE`*.

`Unclaimed` to `Owner`
: This transition occurs when the machine is in use and therefore not available to run jobs. This occurs when the **`START`** expression evaluates to *`FALSE`*.

`Unclaimed` to `Matched`
: This transition occurs when the resource is matched with a job.

`Unclaimed` to `Claimed`
: This transition occurs if **`condor_schedd`** initiates the claiming procedure before the **`condor_startd`** receives notification of the match from the **`condor_negotiator`**.

`Matched` to `Owner`
: This transition occurs if:

- the machine is no longer available to run jobs. This happens when the **`START`** expression evaluates to *`FALSE`*.
- the **`MATCH_TIMEOUT`** timer expires. This occurs when a machine has been matched but not claimed. The machine will eventually give up on the match and become available for a new match.
- **`condor_schedd`** has attempted to claim the machine but encountered an error.
- **`condor_startd`** receives a **`condor_vacate`** command while it is in the `Matched` state.

`Matched` to `Claimed`
: This transition occurs when the machine is successfully claimed and the job is running.

`Claimed` to `Pre-empting`
: From the `Claimed` state, the only possible destination is the `Pre-empting` state. This transition can be caused when:

- The job that has claimed the machine has completed and releases the machine
- The resource is in use. In this case, the **`PREEMPT`** expression evaluates to *`TRUE`*

- **`condor_startd`** receives a **`condor_vacate`** command.
- **`condor_startd`** is instructed to shut down.
- The machine is matched to a job with a higher priority than the currently running job.

`Pre-empting` to *`Claimed`*
: This transition occurs when the resource is matched to a job with a better priority.

`Pre-empting` to *`Owner`*
: This transition occurs when:
  - the **`PREEMPT`** expression evaluated to *`TRUE`* while the machine was in the *`Claimed`* state
  - **`condor_startd`** receives a **`condor_vacate`** command
  - the **`START`** expression evalutes to *`FALSE`* and the job it was running had finished being evicted when it entered the *`Pre-empting`* state.

Machine Activities

While a machine is in a particular state, it will also be performing an *activity*. The possible activities are:

- Idle
- Benchmarking
- Busy
- Suspended
- Retiring
- Vacating
- Killing

Each of these activities has a slightly different meaning, depending on which state they occur in. This list explains each of the possible activities for a machine in different states:

- *`Owner`*
  - *`Idle`*: This is the only possible activity for a machine in the *`Owner`* state. It indicates that the machine is not currently performing any work for MRG Grid, even though it may be working on other unrelated tasks.
- *`Unclaimed`*
  - *`Idle`*: This is the normal activity for machines in the *`Unclaimed`* state. The machine is available to run MRG Grid tasks, but is not currently doing so.
  - *`Benchmarking`*: This activity only occurs in the *`Unclaimed`* state. It indicates that benchmarks are being run to determine the speed of the machine. How often this activity occurs can be adjusted by changing the *`RunBenchmarks`* configuration variable.
- *`Matched`*

  - *`Idle`*: Although the machine is matched, it is still considered *`Idle`*, as it is not currently running a job.

- *`Claimed`*

  - *`Idle`*: The machine has been claimed, but the **`condor_starter`** daemon, and therefore the job, has not yet been started. The machine will briefly return to this state when the job finishes.

  - *`Busy`*: The **`condor_starter`** daemon has started and the job is running.

  - *`Suspended`*: The job has been suspended. The claim is still valid, but the job is not making any progress and MRG Grid is not currently generating a load on the machine.

  - *`Retiring`*: When an active claim is about to be pre-empted, it enters retirement while it waits for the current job to finish. The *`MaxJobRetirementTime`* configuration variable determines how long to wait. Once the job finishes or the retirement time expires, the *`Preempting`* state is entered.

- *`Preempting`*

  - *`Vacating`*: The job that was running is attempting to exit gracefully.

  - *`Killing`*: The machine has requested the currently running job to exit immediately.

9.2. The condor_startd daemon

This section discusses the **`condor_startd`** daemon. This daemon evaluates a number of expressions in order to determine when to transition between states and activities. The most important expressions are explained here.

The **`condor_startd`** daemon represents the machine or slot on which it is running. This daemon is responsible for publishing characteristics about the machine in the machine's ClassAd. To see the values for the attributes, run **`condor_status -l`** ***`hostname`*** from the shell prompt. On a machine with more than one slot, the **`condor_startd`** will regard the machine as separate slots, each with its own name and ClassAd.

Normally, the **`condor_negotiator`** evaluates expressions in the machine ClassAd against job ClassAds to see if there is a match. By locally evaluating an expression, the machine only evaluates the expression against its own ClassAd. If the expression references parameters that can only be found in another ClassAd, then the expression can not be locally evaluated. In this case, the expression will usually evaluate locally to *`UNDEFINED`*.

The START expression

The most important expression to the **`condor_startd`** daemon 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 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 machine locally evaluates the **`IsOwner`** expression to determine if it is capable of running jobs. The default **`IsOwner`** expression is a function of the **`START`** expression, so that **`START =?`**

= FALSE. Any job ClassAd attributes appearing in the **START** expression, and subsequently in the **IsOwner** expression, are undefined in this context, and may lead to unexpected behavior. If the **START** expression is modified to reference job ClassAd attributes, the **IsOwner** expression should also be modified to reference only machine ClassAd attributes.

The REQUIREMENTS expression

The **REQUIREMENTS** expression is used for matching machines with jobs. When a machine is 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.

The RANK expression

A machine can be configured to prefer certain jobs over others, through the use of the **RANK** expression in the machine ClassAd. It can reference any attribute found in either the machine ClassAd or a job ClassAd. The most common use of this expression is to configure a machine so that it prefers to run jobs from the owner of that machine. Similarly, it is often used for a group of machines to prefer jobs submitted by the owners of those machines.

This example demonstrates a simple application of the **RANK** expression

In this example there is a small research group consisting of four machines and four owners:

- The machine called *tenorsax* is owned by the user *coltrane*
- The machine called *piano* is owned by the user *tyner*
- The machine called *bass* is owned by the user *garrison*
- The machine called *drums* is owned by the user *jones*

To implement a policy that gives priority to the machines in this research group, set the **RANK** expression to reference the **Owner** attribute, where it matches one of the people in the group:

```
RANK = Owner == "coltrane" || Owner == "tyner" \
|| Owner == "garrison" || Owner == "jones"
```

Boolean expressions evaluate to either *1* or *0* (*TRUE* or *FALSE*). In this case, if the remote job is owned by one of the preferred users, the **RANK** expression will evaluate to *1*. If the remote job is owned by any other user, it would evaluate to *0*. The **RANK** expression is evaluated as a floating point number, so it will prefer the group users because it evaluates to a higher number.

Example 9.1. A simple application of the **RANK** expression in the machine ClassAd

This example demonstrates a more complex application of the **RANK** expression. It uses the same basic scenario as *Example 9.1, “A simple application of the **RANK** expression in the machine ClassAd”*, but gives the owner a higher priority on their own machine.

This example is on the machine called *bass*, which is owned by the user *garrison*. The following entry would need to be included in a file in the local configuration directory on that machine:

```
RANK = (Owner == "coltrane") + (Owner == "tyner") \
+ ((Owner == "garrison") * 10) + (Owner == "jones")
```

The parentheses in this expression are essential, because the + operator has higher default precedence than ==. Using + instead of `||` allows the system to match some terms and not others.

If a user not in the research group is running a job on the machine called *bass*, the **RANK** expression will evaluate to *0*, as all of the boolean terms evaluate to *0*. If the user *jones* submits a job, his job would match this machine and the **RANK** expression will evaluate to *1*. Therefore, the the job submitted by *jones* would pre-empt the running job. If the user *garrison* (the owner of the machine) later submits a job, the **RANK** expression will evaluate to *10* because the boolean that matches Jimmy gets multiplied by 10. In this case, the job submitted by *garrison* will pre-empt the job submitted by *jones*.

Example 9.2. A more complex application of the **RANK** expression in the machine ClassAd

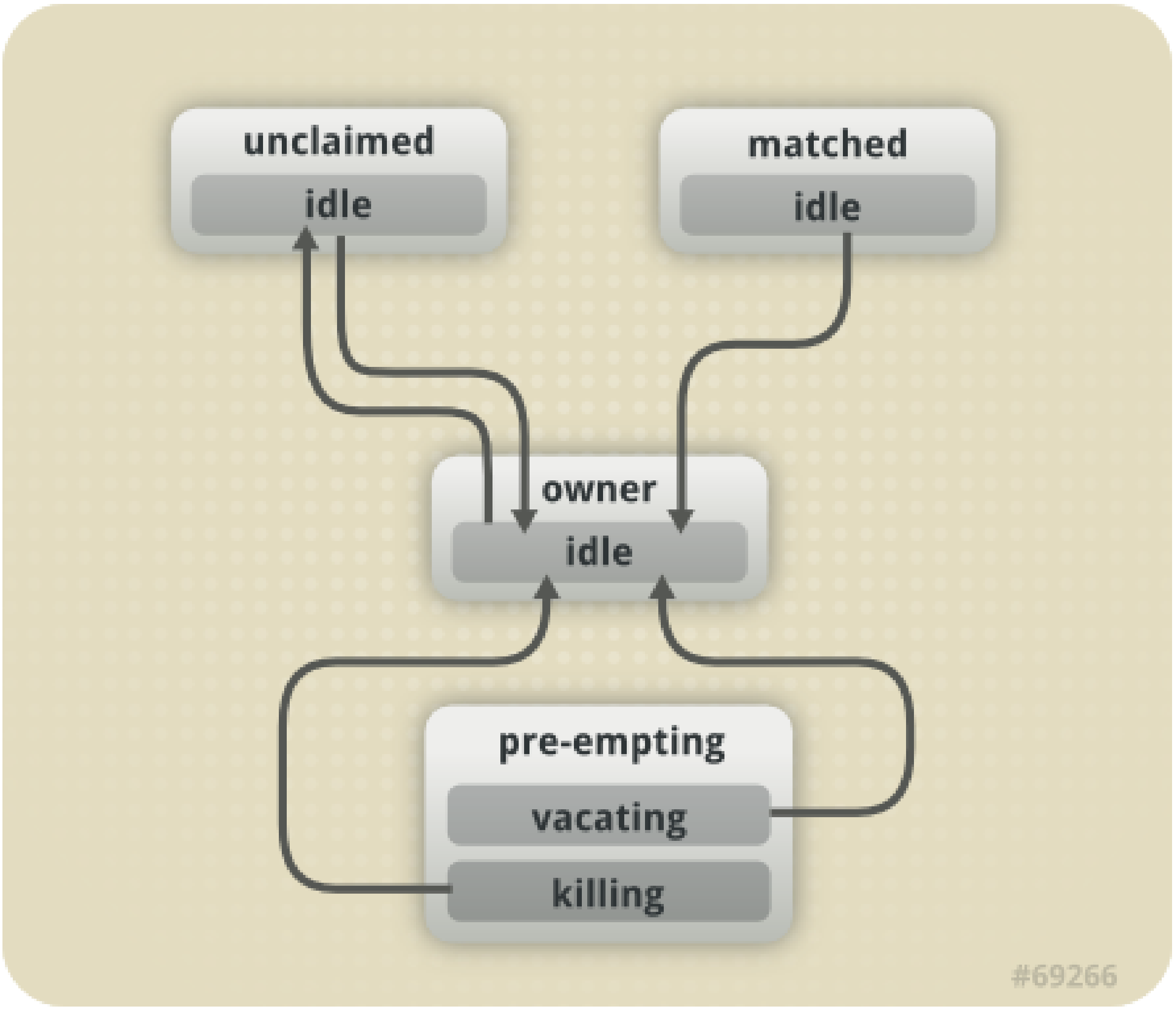
The **RANK** expression can reference parameters other than **Owner**. If one machine has an enormous amount of memory and other do not have much at all, the **RANK** expression can be used to run jobs with larger memory requirements on the machine best suited to it, by using **RANK = ImageSize**. This preference will always service the largest of the jobs, regardless of which user has submitted them. Alternatively, a user could specify that their own jobs should run in preference to those with the largest **ImageSize** by using **RANK = (Owner == "*user_name*" * 1000000000000) + Imagesize**.

9.3. Conditions for state and activity transitions

This section lists all the possible state and activity transitions, with descriptions of the conditions under which each transition occurs.

Owner state

The *Owner* state represents a resource that is currently in use and not available to run jobs. When the **startd** is first spawned, the machine will enter the *Owner* state. The machine remains in the *Owner* state while the **IsOwner** expression evaluates to *TRUE*. If the **IsOwner** expression is *FALSE*, then the machine will transition to *Unclaimed*, indicating that it is ready to begin accepting jobs.



On a shared resource, the default value for the **IsOwner** is optimized to **START =?= FALSE**. This causes the machine to remain in the *Owner* state as long as the **START** expression locally evaluates to *FALSE*. If the **START** expression locally evaluates to *TRUE* or cannot be locally evaluated (in which case, it will evaluate to *UNDEFINED*), the machine will transition to the *Unclaimed* state. For dedicated resources, the recommended value for the **IsOwner** expression is *FALSE*.

Note

The **IsOwner** expression should not reference job ClassAd attributes as this would result in an evaluation of *UNDEFINED*.

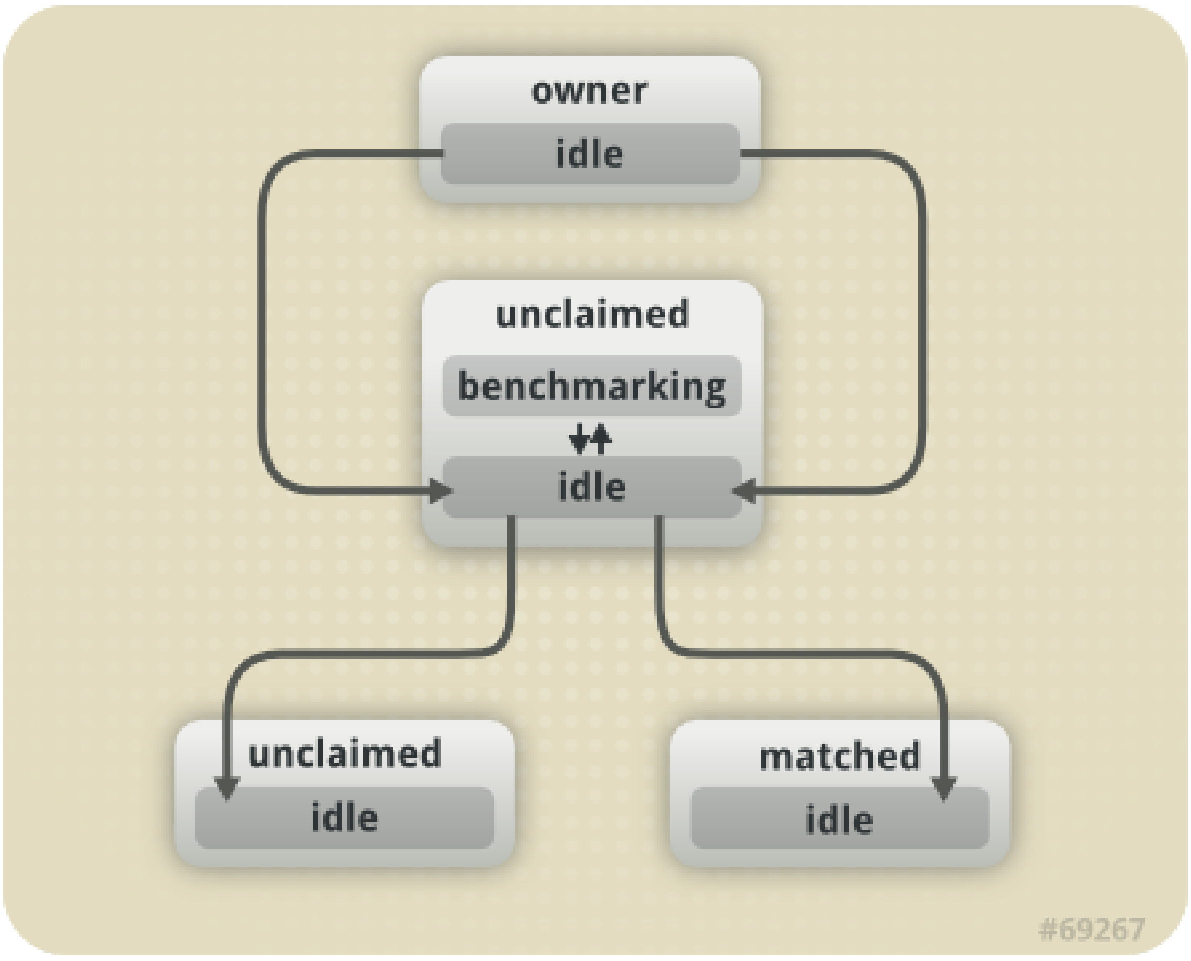
While in the *Owner* state, the **startd** polls the status of the machine. The frequency of this is determined by the **UPDATE_INTERVAL** configuration variable. The poll performs the following actions:

- Calculates load average
- Checks the access time on files
- Calculates the free swap space
- Notes if the **startd** has any critical tasks that need to be performed when the machine moves out of the *Owner* state

Whenever the machine is not actively running a job, it will transition back to the *`Owner`* state. Once a job is started, the value of **`IsOwner`** is no longer relevant and the job will either run to completion or be preempted.

`Unclaimed` state

The *`Unclaimed`* state represents a resource that is not currently in use by its owner or by MRG Grid.



Possible transitions from the *`Unclaimed`* state are:

1. *`Owner:Idle`*
2. *`Matched:Idle`*
3. *`Claimed:Idle`*

When the **`condor_negotiator`** matches a machine with a job, it sends a notification of the match to each. Normally, the machine will enter the *`Matched`* state before progressing to *`Claimed:Idle`*. However, if the job receives the notification and initiates the claiming procedure before the machine receives the notification, the machine will transition directly to the *`Claimed:Idle`* state.

As long as the **`IsOwner`** expression is TRUE, the machine is in the *`Owner`* State. When the **`IsOwner`** expression is *FALSE*, the machine goes into the *`Unclaimed`* state. If the **`IsOwner`** expression is not present in the configuration files, then the default value is **`START =?= FALSE`**. This causes the machine to transition to the *`Owner`* state when the **`START`** expression locally evaluates to *TRUE*.

Effectively, there is very little difference between the *`Owner`* and *`Unclaimed`* states. The most obvious difference is how the resources are displayed in **`condor_status`** and other reporting tools. The only other difference is that benchmarking will only be run on a resource that is in the *`Unclaimed`* state. Whether or not benchmarking is run is determined by the **`RunBenchmarks`** expression. 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. Benchmarking performs and records two performance measures:

- MIPS (Millions of Instructions Per Second); and
- KFLOPS (thousands of FLoating-point Operations Per Second).

When the benchmark is complete the machine returns to *`Idle`*.

This example runs benchmarking every four hours while the machine is in the *`Unclaimed`* state.

A macro called **`BenchmarkTimer`** is used in this example, which records the time since the last benchmark. When this time exceeds four hours, the benchmarks will be run again. A weighted average is used, so the more frequently the benchmarks run, the more accurate the data will be.

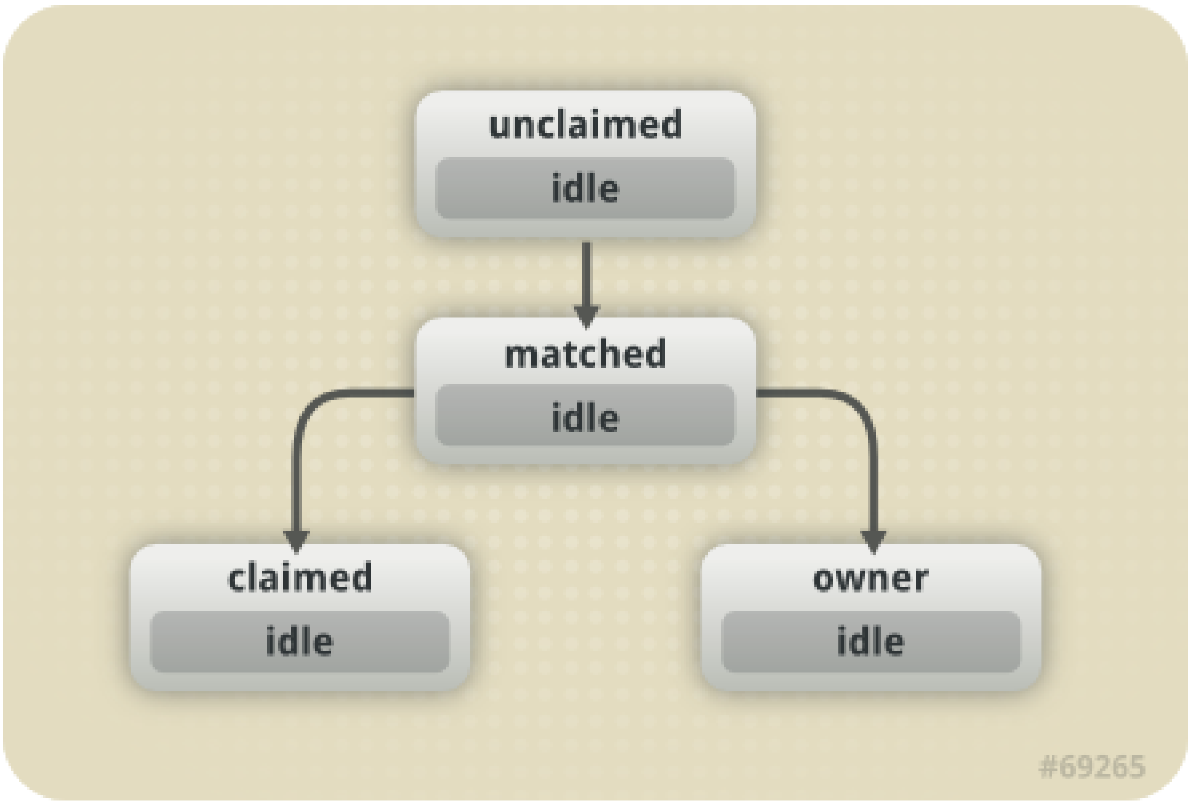
```
BenchmarkTimer = (CurrentTime - LastBenchmark)
RunBenchmarks = $(BenchmarkTimer) >= (4 * $(HOUR))
```

Example 9.3. Setting benchmarks in the machine ClassAd

If **`RunBenchmarks`** is defined and set to anything other than *`FALSE`*, benchmarking will be run as soon as the machine transitions into the *`Unclaimed`* state. To completely disable benchmarks, set **`RunBenchmarks`** to *`FALSE`* or remove it from the configuration file.

`Matched` state

The *`Matched`* state occurs when the machine has been matched to a job by the negotiator, but the job has not yet claimed the machine. Machines are in this state for a very short period before transitioning.



When the machine is matched to a job, the machine will transition into the `Claimed:Idle` state. At any time while the machine is in the `Matched` state, if the **START** expression locally evaluates to *FALSE* the machine will enter the `Owner` state.

Machines in the `Matched` state will adjust the **START** expression so that the requirements evaluate to *FALSE*. This is to prevent the machine being matched again before it has been claimed.

The **startd** will start a timer when a machine transitions into the `Matched` state. This is to prevent the machine from staying in the `Matched` state for too long. The length of the timer can be adjusted with the **MATCH_TIMEOUT** configuration variable, which defaults to 120 seconds (2 minutes). If the job that was matched with the machine does not claim it within this period of time, the machine gives up, and transitions back into the `Owner` state. Normally, it would then transition straight back to the `Unclaimed` state to wait for a new match.

Claimed state

The `Claimed` state occurs when the machine has been claimed by a job. It is the most complex state, with the most possible transitions.

When the machine first enters the `Claimed` state it is in the `Idle` activity. If a job has claimed the machine and the claim will be activated, the machine will transition into the *Busy* activity and the job started. If a **condor_vacate** arrives, or the **START** expression locally evaluates to *FALSE*, it will enter the `Retiring` activity before transitioning to the `Pre-empting` state.

While in `Claimed:Busy`, the **startd** daemon will evaluate the **WANT_SUSPEND** expression to determine which other expression to evaluate. If **WANT_SUSPEND** evaluates to *TRUE*, the **startd** will evaluate the **SUSPEND** expression to determine whether or not to transition to `Claimed:Suspended`. Alternatively, if **WANT_SUSPEND** evaluates to *FALSE* the **startd** will evaluate the **PREEMPT** expression to determine whether or not to skip the `Suspended` state and move to `Claimed:Retiring` before transitioning to the the `Preempting` state.

While a machine is in the *Claimed* state, the **startd** daemon will poll the machine for a change in state much more frequently than while in other states. The frequency can be adjusted by changing the **POLLING_INTERVAL** configuration variable.

The **condor_vacate** command affects the machine when it is in the *Claimed* state. There are a variety of events that may cause the **startd** daemon to try to suspend or kill a running job. Possible causes could be:

- The owner has resumed using the machine
- Load from other jobs
- The **startd** has been instructed to shut down
- The appearance of a higher priority claim to the machine by a different MRG Grid user.

The **startd** can be configured to handle interruptions in different ways. Activity on the machine could be ignored, or it could cause the job to be suspended or even killed. Desktop machines can benefit from a configuration that goes through successively more dramatic actions to remove the job. The least costly option to the job is to suspend it. If the owner is still using the machine after suspending the job for a short while, then **startd** will attempt to vacate the job. Vanilla jobs are sent a soft kill signal, such as **SIGTERM**, so that they can gracefully shut down. If the owner wants to resume using the machine, and vacating can not be completed, the **startd** will progress to kill the job. Killing is a quick death to a job. It uses a hard-kill signal that cannot be intercepted by the application. For vanilla jobs, vacating and killing are equivalent actions.

Pre-empting state

The *Pre-empting* state is used to evict a running job from a machine, so that a new job can be started. There are two possible activities while in the *Pre-empting* state. Which activity the machine is in is dependent on the value of the **WANT_VACATE** expression. If **WANT_VACATE** evaluates to *TRUE*, the machine will enter the *Vacating* activity. Alternatively, if **WANT_VACATE** evaluates to *FALSE*, the machine will enter the *Killing* activity.

The main function of the *Pre-empting* state is to remove the **condor_starter** associated with the job. If the **condor_starter** associated with a given claim exits while the machine is still in the *Vacating* activity, then the job has successfully completed a graceful shutdown, and the application was given the opportunity to intercept the soft kill signal.

While in the *Pre-empting* state the machine advertises its **Requirements** expression as *FALSE*, to signify that it is not available for further matches. This is because it is about to transition to the *Owner* state, or because it has already been matched with a job that is currently pre-empting and further matches are not allowed until the machine has been claimed by the new job.

While the machine is in the *Vacating* activity, it continually evaluates the **KILL** expression. As soon as it evaluates to *TRUE*, the machine will enter the *Killing* activity.

When the machine enters the *Killing* activity it attempts to force the **condor_starter** to immediately kill the job. Once the machine has begun to kill the job, the **condor_startd** starts a timer. The length of the timer defaults to 30 seconds and can be adjusted by changing the **KILLING_TIMEOUT** macro. If the timer expires and the machine is still in the *Killing* activity, it is assumed that an error has occured with the **condor_starter** and the startd will try to vacate the job immediately by sending **SIGKILL** to all of the children of the **condor_starter** and then to the **condor_starter** itself.

Once the **`condor_starter`** has killed 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 *`Killing`* activity. If the job was pre-empted because a better match was found, the machine will enter *`Claimed:Idle`*. If the pre-emption was caused by the machine owner, the machine will enter the *`Owner`* state.

9.4. Defining a policy

When a transition occurs, MRG Grid records the time that the new activity or state was entered. These times can be used to write expressions for customized transitions. To define a policy, set expressions in the configuration file (see *Chapter 2, Configuration* for an introduction to configuration). 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.

Warning

If you intend to change any of the settings as described in this chapter, make sure you follow the instructions carefully and always test your changes before implementing them. Mistakes in policy configuration can have a severe negative impact on both the owners of machines in your pool, and the users who submit jobs to those machines.

Default macros

The following default macros assist with writing human-readable expressions.

`MINUTE`
: Defaults to *`60`*

`HOUR`
: Defaults to *`(60 * $(MINUTE))`*

`StateTimer`
: Amount of time in the current state

  Defaults to *`(CurrentTime - EnteredCurrentState)`*

`ActivityTimer`
: Amount of time in the current activity

  Defaults to *`(CurrentTime - EnteredCurrentActivity)`*

`ActivationTimer`
: Amount of time the job has been running on this machine

  Defaults to *`(CurrentTime - JobStart)`*

`NonCondorLoadAvg`
: The difference between the system load and the MRG Grid load (equates to the load generated by everything except MRG Grid)

  Defaults to *`(LoadAvg - CondorLoadAvg)`*

`BackgroundLoad`
Amount of background load permitted on the machine and still be able to start a job

Defaults to *`0.3`*

`HighLoad`
If the **`NonCondorLoadAvg`** goes over this, the CPU is considered too busy, and eviction of the job should start

Defaults to *`0.5`*

`StartIdleTime`
Amount of time the keyboard must be idle before starting a job

Defaults to *`15 * $(MINUTE)`*

`ContinueIdleTime`
Amount of time the keyboard must to be idle before resumption of a suspended job

Defaults to *`5 * $(MINUTE)`*

`MaxSuspendTime`
Amount of time a job may be suspended before more drastic measures are taken

Defaults to *`10 * $(MINUTE)`*

`MaxVacateTime`
Amount of time a job may spend attempting to exit gracefully before giving up and being killed

Defaults to *`10 * $(MINUTE)`*

`KeyboardBusy`
A boolean expression that evaluates to *`TRUE`* when the keyboard is being used

Defaults to *`KeyboardIdle < $(MINUTE)`*

`CPUIdle`
A boolean expression that evaluates to *`TRUE`* when the CPU is idle

Defaults to *`$(NonCondorLoadAvg) <= $(BackgroundLoad)`*

`CPUBusy`
A boolean expression that evaluates to *`TRUE`* when the CPU is busy

Defaults to *`$(NonCondorLoadAvg) >= $(HighLoad)`*

`MachineBusy`
The CPU or the Keyboard is busy

Defaults to *`($(CPUBusy) || $(KeyboardBusy)`*

`CPUIsBusy`
A boolean value set to the same value as **`CPUBusy`**

`CPUBusyTime`
: the time in seconds since **`CPUBusy`** became *TRUE*. Evaluates to *`0`* if **`CPUBusy`** is *FALSE*

It is preferable to suspend jobs instead of killing them. This is especially true when the job uses little memory, when the keyboard is not being used or when the job is running in the vanilla universe. By default, these macros will gracefully vacate jobs that have been running for more than ten minutes, or are vanilla universe jobs:

```
WANT_SUSPEND        = ( $(SmallJob) || $(KeyboardNotBusy) || $(IsVanilla) )
WANT_VACATE         = ( $(ActivationTimer) > 10 * $(MINUTE) || $(IsVanilla) )
```

Expression Definitions

This list gives examples of typical expressions.

`START`

Start a job if the keyboard has been idle long enough and the load average is low enough or if the machine is currently running a job. Note that MRG Grid will only run one job at a time, but it may pre-empt the currently running job in favour of the new one:

```
START   = ( (KeyboardIdle > $(StartIdleTime)) \
  && ( $(CPUIdle) || (State != "Unclaimed" \
  && State != "Owner")) )
```

`SUSPEND`

Suspend a job if the keyboard is in use. 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`

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)) )
```

`PREEMPT`

There are two conditions that signal pre-emption. 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)) )
```

`MaxJobRetirementTime`

Do not give jobs any time to retire on their own when they are about to be pre-empted:

```
MaxJobRetirementTime = 0
```

KILL

Kill jobs that take too long to exit gracefully:

```
KILL   = $(ActivityTimer) > $(MaxVacateTime)
```

Example Policies

The following examples show how to use the default macros detailed in this chapter to create commonly used policies.

This example shows how to set up a machine for running test jobs from a specified user.

The machine needs to behave normally unless the user *coltrane* submits a job. When this occurs, the job should start execution immediately, regardless of what else is happening on the machine at that time.

Jobs submitted by *coltrane* should not be suspended, vacated or killed. This is reasonable because *coltrane* will only be submitting very short running programs for testing purposes.

```
START       = ($(START)) || Owner == "coltrane"
SUSPEND     = ($(SUSPEND)) && Owner != "coltrane"
CONTINUE    = $(CONTINUE)
PREEMPT     = ($(PREEMPT)) && Owner != "coltrane"
KILL        = $(KILL)
```

There are no specific settings for the **CONTINUE** or **KILL** expressions. Because the jobs submitted by *coltrane* will never be suspended, the **CONTINUE** expression is irrelevant. Similarly, because the jobs can not be pre-empted, **KILL** is irrelevant.

Example 9.4. Test-job Policy

This example shows how to set up a machine to only run jobs at certain times of the day.

This is achieved through the **ClockMin** and **ClockDay** attributes. These are special attributes which are automatically inserted by the **condor_startd** into its ClassAd, so they can always be referenced in policy expressions. **ClockMin** defines the number of minutes that have passed since midnight. **ClockDay** defines the day of the week, where Sunday = 0, Monday = 1, and so on to Saturday = 7.

To make the policy expressions easier to read, use macros to define the time periods when you want jobs to run or not run. Regular work hours at your site could be defined as being from 0800 until 1700, Monday through Friday.

```
WorkHours = ( (ClockMin >= 480 && ClockMin < 1020) && \
  (ClockDay > 0 && ClockDay < 6) )
AfterHours = ( (ClockMin < 480 || ClockMin >= 1020) || \
  (ClockDay == 0 || ClockDay == 6) )
```

Once these macros are defined, MRG Grid can be instructed to 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) )
```

Avoid suspending jobs during work hours so that in the morning, if a job is running, it will be immediately pre-empted instead of being suspended for some length of time:

```
WANT_SUSPEND = $(AfterHours)
```

By default, the **`MachineBusy`** macro is used to define the **`SUSPEND`** and **`PREEMPT`** expressions. If you have changed these expressions, you will need to add **`$(WorkHours)`** to your **`SUSPEND`** and **`PREEMPT`** expressions as appropriate.

Example 9.5. Time of Day Policy

This example shows how to set up a pool of machines that include desktop machines and dedicated cluster machines, requiring different policies.

In this scenario, keyboard activity should not have any effect on the dedicated machines. It might be necessary to log into the dedicated machines to debug a problem, or change settings, and this should not interrupt the running jobs. 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 create a standard desktop policy and a standard non-desktop policy. The appropriate policy is then stored in a file located in the local configuration directory for each machine. This example, however, defines one standard policy in **`condor_config`** with a toggle that can be set in the local configuration directory.

If **`IsDesktop`** is configured, make it an attribute of the machine ClassAd:

```
STARTD_EXPRS = IsDesktop
```

If a desktop machine, only consider starting jobs if the load average is low enough or the machine is currently running a Condor job, and the user is not active:

```
START = ( ($(CPUIdle) || (State != "Unclaimed" && State != "Owner")) \
 && (IsDesktop =!= True || (KeyboardIdle > $(StartIdleTime))) )
```

Suspend instead of vacating or killing for small or vanilla universe jobs:

```
WANT_SUSPEND = ( $(SmallJob) || $(JustCpu) \
  || $(IsVanilla) )
```

When pre-empting, vacate instead of killing for jobs that have been running for longer than 10 minutes, or vanilla universe jobs:

```
WANT_VACATE  = ( $(ActivationTimer) > 10 * $(MINUTE) \
  || $(IsVanilla) )
```

Suspend jobs if the CPU has been busy for more than 2 minutes and the job has been running for more than 90 seconds. Also suspend jobs if this is a desktop and the user is active:

```
SUSPEND = ( ((CpuBusyTime > 2 * $(MINUTE)) && ($(ActivationTimer) > 90)) \
 || ( IsDesktop =?= True && $(KeyboardBusy) ) )
```

Continue jobs on a desktop machine if the CPU is idle, the job has been suspended more than 5 minutes and the keyboard has been idle for long enough:

```
CONTINUE = ( $(CPUIdle) && ($(ActivityTimer) > 300) \
 && (IsDesktop =!= True || (KeyboardIdle > $(ContinueIdleTime))) )
```

Pre-empt jobs if it has been suspended too long or the conditions to suspend the job has been met, but suspension is not desired:

```
PREEMPT = ( ((Activity == "Suspended") && \
 ($(ActivityTimer) > $(MaxSuspendTime))) \
 || (SUSPEND && (WANT_SUSPEND == False)) )
```

The following expression determines retirement time. Replace *0* with the desired amount of retirement time for dedicated machines. 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 directories for desktops can now be configured with the following line:

```
IsDesktop = True
```

In all other cases, the default policy described above will ignore keyboard activity.

Example 9.6. Desktop/Non-Desktop Policy

This example shows how to prevent and disable pre-emption.

Pre-emption can result in jobs being killed. When this happens, the jobs remain in the queue and will be automatically rescheduled. It is highly recommend designing jobs that work well in this environment, rather than simply disabling pre-emption. Planning for pre-emption makes jobs more robust in the face of other sources of failure. The easiest way to do this is to use the standard universe. If a job is incompatible with the requirements of the standard universe, the job can still gracefully shutdown and restart by intercepting the soft kill signal.

However, there can be cases where it is appropriate to force MRG Grid to never kill jobs within an upper time limit. This can be achieved with the following policy.

Allow a job to run uninterrupted for up to two days before forcing it to vacate:

```
MAXJOBRETIREMENTTIME = $(HOUR) * 24 * 2
```

Construction of this expression can be more complex. For example, it could specify a different retirement time for different users or different types of jobs. Additionally, the job might come with its own definition of **`MAXJOBRETIREMENTTIME`**, but this can 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 could decrease the number of cycles that are wasted on jobs that are killed.

Note that the use of **`MAXJOBRETIREMENTTIME`** limits the killing of jobs, but it does not prevent the pre-emption of resource claims. Therefore, it is technically not a way of disabling pre-emption, but simply a way of forcing pre-empting claims to wait until an existing job finishes or runs out of time.

To limit the pre-emption of resource claims, the following policy can 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 pre-emption by machine activity:

```
PREEMPT = False
```

Disable pre-emption by user priority:

```
PREEMPTION_REQUIREMENTS = False
```

Disable pre-emption by machine rank by ranking all jobs equally:

```
RANK = 0
```

When disabling claim pre-emption, it is advised to also optimize negotiation:

```
NEGOTIATOR_CONSIDER_PREEMPTION = False
```

Without any pre-emption 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. To avoid this behavior, force claims to be retired after a specified period of time bys etting the **`CLAIM_WORKLIFE`** variable. This enforces a time limit, beyond which no new jobs may be started on an existing claim. In this case, the **`condor_schedd`** daemon is forced to go back to the **`condor_negotiator`** to request a new match. The execute machine configuration would include a line that forces the schedd to renegotiate for new jobs after 20 minutes:

```
CLAIM_WORKLIFE = 1200
```

It is not advisable to set **`NEGOTIATOR_CONSIDER_PREEMPTION`** to *`False`*, as it can potentially lead to some machines never being matched to jobs.

Example 9.7. Disabling Pre-emption

This example shows how to create a policy around job suspension.

When jobs with a higher priority are submitted, the executing jobs might be pre-empted. These jobs can 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.

A policy can be created that will allow jobs to be suspended instead instead of evicted. The policy utilizes two slots: *`slot1`* only runs jobs identified as high priority jobs; *`slot2`* is set to run jobs according to the usual policy and to suspend them when *`slot1`* is claimed. A policy for a machine with more than one physical CPU may be adapted from this example. Instead of having two slots, you would have twice times the number of physical CPUs. Half of the slots would be for high priority jobs and the other half would be for suspendable jobs.

Tell MRG Grid that the machine has two slots, even though it only has a single CPU:

```
NUM_CPUS = 2
```

`slot1` is the high-priority slot, while *`slot2`* is the background slot:

```
START = (SlotID == 1) && $(SLOT1_START) || \
 (SlotID == 2) && $(SLOT2_START)
```

Only start jobs on *`slot1`* if the job is marked as a high-priority job:

```
SLOT1_START = (TARGET.IsHighPrioJob =?= TRUE)
```

Only start jobs on *`slot2`* if there is no job on *`slot1`*, and if the machine is otherwise idle. Note that the *`Busy`* activity is only in the *`Claimed`* state, and only when there is an active job:

```
SLOT2_START = ( (slot1_Activity != "Busy") && \
 (KeyboardIdle > $(StartIdleTime)) && \
 ($(CPUIdle) || (State != "Unclaimed" && State != "Owner")) )
```

Suspend jobs on *`slot2`* if there is keyboard activity or if a job starts on *`slot1`*:

```
SUSPEND = (SlotID == 2) && \
 ( (slot1_Activity == "Busy") || ($(KeyboardBusy)) )

CONTINUE = (SlotID == 2) && \
 (KeyboardIdle > $(ContinueIdleTime)) && \
 (slot1_Activity != "Busy")
```

Note that in this example, the job ClassAd attribute **`IsHighPrioJob`** has no special meaning. It is an invented name chosen for this example. To take advantage of the policy, a user must submit high priority jobs with this attribute defined. The following line appears in the job's submit description file as:

```
+IsHighPrioJob = True
```

Example 9.8. Job Suspension

User Priorities and Negotiation

MRG Grid uses priorities and negotiation to allocate jobs between the machines in the pool. When a job is advertised to the pool, it is ranked according to the user that submitted it. High-priority users will get their jobs run before low-priority users.

Every user is identified by *`username@uid_domain`* and is assigned a priority value. The priority value is assigned directly to the username and domain, so the same user can submit jobs from multiple machines. The highest possible priority is 1, and the priority decreases as the number rises. There are two priority values assigned to users:

- *Real user priority* (RUP), which measures the amount of resources consumed by the user.
- *Effective user priority* (EUP), which determines the number of resources available to the user.

Real User Priority (RUP)

RUP measures the amount of resources consumed by the user over time. Every user begins with a RUP of 0.5 and will stabilize over time if the user consumes resources at a stable rate. For example, if a user continuously uses exactly ten resources for a long period of time, the RUP of that user will stabilize to 10.

The RUP will get better as the user decreases the amount of resources being consumed. The rate at which the RUP decays can be set in the configuration files using the **`PRIORITY_HALFLIFE`** setting, which measures in seconds. For example, if the **`PRIORITY_HALFLIFE`** is set to 86400 (1 day), and a user who's RUP is 10 removes all their jobs and consumes no further resources, the RUP would become 5 in one day, 2.5 in two days, and so on.

Effective User Priority (EUP)

EUP is used to determine how many resources a user can access. The EUP is related to the RUP by a priority factor which can be defined on a per-user basis. By default, the priority factor for all users is 1.0, and so the EUP will remain the same as the the RUP. This can be used to preferentially serve some users over others.

The number of resources that a user can access is inversely related to the EUP of each user. For example, Alice has an EUP of 5, Bob has an EUP of 10 and Charlie has an EUP of 20. In this case, Alice will be able to access twice as many resources as Bob, who can access twice as many as Charlie. However, if a user does not consume the full amount of resources they have been allocated, the remainder will be redistributed among the remaining users.

There are two settings that can affect EUP when submitting jobs:

Nice users
: A *nice user* gets their RUP raised by a priority factor, which is specified in the configuration file. This results in a large EUP and subsequently a low priority for access to resources, causing the job to run as the equivalent of a background job.

Remote Users
: In some situations, users from other domains may be able to submit jobs to the local pool. It may be preferable to treat local users preferentially over remote users. In this case, a *remote user* would get their RUP raised by a priority factor, which is specified in the configuration file. This results in a large EUP and subsequently a low priority for access to resources.

Pre-emption

Priorities are used to ensure that users get an appropriate allocation of resources. MRG Grid can also pre-empt jobs and reallocate them if conditions change, so that higher priority jobs are continually pushed further up the queue.

However, too many pre-emptions can lead to a condition known as *thrashing*, where a new job with a higher priority is identified every cycle. In this situation, every job gets pre-empted and no job has a chance to finish. To avoid thrashing, conditions for pre-emption can be set using the **`PREEMPTION_REQUIREMENTS`** setting in the configuration file. Set this variable to deny pre-emption when the current job has been running for a relatively short period of time. This limits the number of pre-emptions per resource, per time period. There is more information about the **`PREEMPTION_REQUIREMENTS`** setting in *Chapter 2, Configuration*.

Negotiation

MRG Grid uses negotiation to match jobs with the resources capable of running them. The **`condor_negotiator`** daemon is responsible for negotiation.

Negotiation occurs in cycles. During a negotiation cycle, the **`condor_negotiator`** daemon performs the following actions, in this order:

1. Construct a list of all possible resources in the pool

2. Obtain a list of all job submitters in the pool

3. Sort the list of job submitters based on EUP, with the highest priority user (lowest EUP) at the top of the list, and the lowest at the bottom.

4. Continue to perform all four steps until there are either no more resources to match, or no more jobs to match.

Once the **`condor_negotiator`** daemon has finished the initial actions, it will list every job for each submitter, in EUP order. Since jobs can be submitted from more than one machine, there is further sorting. When the jobs all come from a single machine, they are sorted in order of job priority. Otherwise, all the jobs from a single machine are sorted before sorting the jobs from the next machine.

In order to find matches, **`condor_negotiator`** will perform the following tasks for each machine in the pool that can execute jobs:

1. If *`machine.requirements`* is false or *`job.requirements`* is false, ignore the machine

2. If the machine is in the *`Claimed`* state, but not running a job, ignore the machine

3. If the machine is not running a job, add it to the potential match list with a reason of *`No Preemption`*

4. If the machine is running a job:

   a. If the *`machine.RANK`* on the submitted job is higher than that of the running job, add this machine to the potential match list with a reason of *`Rank`*

   b. If the EUP of the submitted job is better than the EUP of the running job, **`PREEMPTION_REQUIREMENTS`** is true, and the *`machine.RANK`* on the submitted job is higher than the running job, add this machine to the potential match list with a reason of *`Priority`*

The potential match list is sorted by:

1. *NEGOTIATOR_PRE_JOB_RANK*
2. *job.RANK*
3. *NEGOTIATOR_POST_JOB_RANK*
4. Reason for claim
   - *No Preemption*
   - *Rank*
   - *Priority*
5. *PREEMPTION_RANK*
6. Order in queue

The job is then assigned to the top machine on the potential match list. That machine is then removed from the list of resources available in this negotiation cycle and the daemon goes on to find a match for the next job.

Cluster Considerations

If a cluster has multiple jobs and one of them cannot be matched, no other jobs in that cluster will be returned during the current negotiation cycle. This is based on an assumption that all the jobs in a cluster will be similar. The configuration variable **NEGOTIATE_ALL_JOBS_IN_CLUSTER** can be used to disable this behaviour. The definition of what makes up a cluster can be modified by use of the **SIGNIFICANT_ATTRIBUTES** setting.

Group Accounting

MRG Grid keeps a running tally of resource use. This accounting information is used to calculate priorities for the scheduling algorithms. Accounting is done on a per-user basis by default, but can also be on a per-group basis. When done on a per-group basis, any jobs submitted by the same group will be treated with the same priority.

When a job is submitted, the user can include an attribute that defines the accounting group. For example, the following line in a job's submit description file indicates that the job is part of the *group_physics* accounting group:

```
+AccountingGroup = "group_physics"
```

Example 10.1. Submit description file entry when using accounting groups

The value for the *AccountingGroup* attribute is a string. It must be enclosed in double quotation marks and can contain a maximum of 40 characters. The name should not be qualified with a domain, as parts of the system will add the **$(UID_DOMAIN)** to the string. For example, the statistics for this accounting group might be displayed as follows:

```
User Name                                EUP
------------------------------     ---------
group_physics@example.com               0.50
mcurie@example.com                     23.11
```

```
pvonlenard@example.com                  111.13
...
```

Example 10.2. Accounting group statistics, showing the appending of the fully qualified domain

Condor normally removes entities automatically when they are no longer relevant, however administrators can also remove accounting groups manually, using the *`-delete`* option with the **`condor_userprio`** daemon. This action will only work if all jobs have already been removed from the accounting group, and the group is identified by its fully-qualified name. For example:

```
$ condor_userprio -delete group_physics@example.com
```

Example 10.3. Manually removing accounting groups

10.1. Group Quotas

In some cases, priorities based on each individual user might not be effective. *Group quotas* affect the negotiation for available resources within the pool. This may be the case when different groups own different amounts of resources, and the groups choose to combine their resources to form a pool. For example:

The physics department owns twenty workstations, and the chemistry department owns ten workstations. They have combined their resources to form a pool of thirty similar machines. The physics department wants priority on any twenty of the workstations. Likewise, the chemistry department wants priority on any ten workstations.

By creating group quotas, users are allocated not to specific machines, but to numbers of machines (a quota). Given thirty similar machines, group quotas allow the 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 currently available.

Example 10.4. An effective use of group quotas

In order to set group quotas, the group must be identified in the job's submit description file, using the *`AccountingGroup`* attribute. Members of a group quota are called *group users*. When specifying a group user, you will need to include the name of the group, as well the username, using the following syntax:

```
+AccountingGroup = "group.user"
```

For example, if the user mcurie of the *`group_physics`* group was submitting a job in a pool that implements group quotas, the submit description file would be:

```
+AccountingGroup = "group_physics.mcurie"
```

Example 10.5. Submit description file entry when using group quotas

Group names are not case-sensitive and do not require the *`group_`* prefix. However, group names should be unique, to avoid conflicts. Adding the *`group_`* prefix to group names ensures against conflicts.

Quotas are configured in terms of slots per group. The combined quotas for all groups must be equal to or less than the amount of available slots in the pool. Any slots that are not allocated as part of a group quota are allocated to the *none* group. The *none* group contains only those users who do not submit jobs as part of a group.

Changes caused by group quotas to accounting and negotiation

When using group quotas, some changes occur in how accounting and negotiation are processed.

For jobs submitted by group users, accounting is performed per group user, rather than per group or individual user.

Negotiation is performed differently when group quotas are used. Instead of negotiating in the order described in *Negotiation*, the **condor_negotiator** daemon will create a list of all jobs belonging to defined groups before it lists those jobs submitted by individual submitters. If there is more than one group in the negotiation cycle, the daemon will negotiate for the group using the smallest percentage of resources first, and the highest percentage last. However, the same algorithm still applies to individual submitters.

Managing configuration for group quotas

Configuring a pool can be slightly different when using group quotas. Each group can be assigned an initial value for user priority with the **GROUP_PRIO_FACTOR_** setting. Additionally, if a group is currently allocated the entire quota of machines, and a group user has a submitted job that is not running, the **GROUP_AUTOREGROUP_** setting, if true, will allow the job to considered again within the same negotiation cycle, along with the individual users jobs.

- **GROUP_NAMES = group_physics, group_chemistry**
- **GROUP_QUOTA_group_physics = 20**
- **GROUP_QUOTA_group_chemistry = 10**
- **GROUP_PRIO_FACTOR_group_physics = 1.0**
- **GROUP_PRIO_FACTOR_group_chemistry = 3.0**
- **GROUP_AUTOREGROUP_group_physics = FALSE**
- **GROUP_AUTOREGROUP_group_chemistry = TRUE**

In this example, the physics group can access 20 machines and the chemistry group can access ten machines. The initial priority factor for users within the groups are 1.0 for the physics group and 3.0 for the chemistry group. The **GROUP_AUTOREGROUP_** settings indicate that the physics group will never be able to access more than 20 machines, while the chemistry group could potentially get more than ten machines.

Example 10.6. Example configuration for group quotas

10.2. Job Priorities

Job Priority

In addition to user priorities, it is also possible to specify job priorities to control the order of job execution. Jobs can be assigned a priority level, of any integer, through the use of the **condor_prio** command. Jobs with a higher number will run with a higher priority. Job priority works only on a per user basis. It is effective when used by a single user to order their own jobs, but will not impact the order in which they run with other jobs in the pool.

1. To find out what jobs are currently running, use the **condor_q** with the name of the user to query:

```
$  condor_q user
-- 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   user           4/11 15:06   0+00:00:00 I  0   0.3  hello

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

2. Job priority can be any integer. The default priority is *0*. To change the priority use the **condor_prio** with the desired priority:

   ```
   $  condor_prio -p -15 126.0
   ```

3. To check that the changes have been made, use the **condor_q** command again:

   ```
   $  condor_q user
   -- 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   user           4/11 15:06   0+00:00:00 I  -15 0.3  hello

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

10.3. Hierarchical Fair Share (HFS)

Hierarchical Fair Share (HFS) is a feature that allows user quotas and priorities to be managed within an administrator-defined hierarchy. MRG Grid provides the ability to specify HFS hierarchies of any depth and breadth.

Using dynamic quotas is preferable to static quotas, due to the dynamic and flexible nature of grid computing. When two or more children with static quotas are declared, if the children quotas add up to more than the parent quota, the available quota will be allocated on a first-come first served basis. Dynamic quotas can help to resolve this problem.

Setting up hierarchical fair share

1. The hierarchical groups must first be specified in the global configuration file. Groups are delimited from subgroups by a period (*.*) symbol, in the same way as groups are delimited from users. If two groups are defined as *group_physics* and *group_chemistry*, the subgroups are defined as *group_physics.lab1*, *group_physics.lab2*, *group_chemistry.lab1*, and *group_chemistry.lab2*. Users submit jobs to these subgroups by adding a plus (+) symbol and the name of the accounting group to the job submit description file. For example, if user *mcurie* wants to submit a job to *group_physics.lab1*, they would add *+Accounting_Group = "group_physics.lab1.mcurie"* to their job submit description file.

   Each group must also have a quota declaration. Quota declarations can be either *dynamic* or *static*. A static quota is expressed as a single integer, representing a specific slot count. If a static quota for a subgroup is declared, and the amount is greater than the quota available to the parent group, the subgroup quota will be reset to the maximum available in the parent group. A dynamic quota is specified as a fractional value between *0.0* and *1.0*. This represents the fraction of the quota allocated to the parent group, that the subgroup is able to use.

   If the total quota of the subgroups within a parent group is less than the total quota for the parent group, the remainder is available for users to submit jobs using the parent accounting group. If the total quota of the subgroups in a parent group equal the total quota for the parent group, the parent group will not have enough quota available for users to submit jobs to the parent group. For example, if *`group_physics`* and *`group_chemistry`* are top level groups and their quota adds up to 0.8, there will be 0.2 quota remaining for users that are not associated with any group. If groups *`group_physics.lab1`* and *`group_physics.lab2`* have quotas that sum to 1.0, then no jobs can be submitted directly to group *`group_physics`*, only to its subgroups.

```
GROUP_NAMES = group_physics, group_chemistry, group_physics.lab1,
group_physics.lab2, group_physics.lab3, group_physics.lab3.team1,
group_physics.lab3.team2, group_physics.lab3.team3,
group_chemistry.lab1, group_chemistry.lab2

GROUP_QUOTA_DYNAMIC_group_physics = .4
GROUP_QUOTA_DYNAMIC_group_chemistry = .4
GROUP_QUOTA_DYNAMIC_group_chemistry.lab1 = .4
GROUP_QUOTA_DYNAMIC_group_chemistry.lab2 = .6
GROUP_QUOTA_DYNAMIC_group_physics.lab1= .2
GROUP_QUOTA_DYNAMIC_group_physics.lab2= .2
GROUP_QUOTA_DYNAMIC_group_physics.lab3= .6
GROUP_QUOTA_DYNAMIC_group_physics.lab3.team1 = .2
GROUP_QUOTA_DYNAMIC_group_physics.lab3.team2 = .2
GROUP_QUOTA_DYNAMIC_group_physics.lab3.team3 = .4
```

2. The **`autoregroup`** feature allows groups to use quota that is unused by other groups. To enable **`autoregroup`** for a group, set the **`GROUP_AUTOREGROUP_`*`group`*** configuration variable to *`TRUE`*. Autoregroup works in a hierarchy. Subgroups with the same parent get first chance at slots that are unused by their subgroup peers. If these slots cannot be used, they are passed up the group hierarchy, as long as **`autoregroup = TRUE`** is enabled at the higher levels:

```
GROUP_AUTOREGROUP_group_physics = TRUE
GROUP_AUTOREGROUP_group_physics.lab3 = TRUE
GROUP_AUTOREGROUP_group_physics.lab3.team1 = TRUE
GROUP_AUTOREGROUP_group_chemistry = TRUE
GROUP_AUTOREGROUP_group_chemistry.lab1 = TRUE
GROUP_AUTOREGROUP_group_chemistry.lab2 = TRUE
```

   This behavior is slightly different in two situations:

   - Where a job is not submitted with *`+AccountingGroup`* specified. These jobs fall into a pool that has **`autoregroup = true`** set. These jobs will not run unless the top level group quotas sum to less than 1.0.
   - Where a job is submitted against a group that has children groups. This pool will exhibit **`autoregroup = true`** behavior. The extent to which these submitters can claim unused slots is determined by the parent group's autoregroup hierarchy.

3. When creating the job submit description file, specify the group and user:

```
executable = /bin/sleep
arguments = 600
universe = vanilla
should_transfer_files = YES
when_to_transfer_output = ON_EXIT
+AccountingGroup = "group_physics.lab3.team1.mcurie"
```

```
queue 100
```

The Virtual Machine Universe

Virtual machines can be operated under MRG Grid using Xen or KVM (with libvirt). MRG Grid requires some configuration before being used with virtual machines. This chapter contains information on getting started.

Before configuring MRG Grid to work with virtual machines, install the virtualization package according to the vendor's instructions.

In order for MRG Grid to fully support virtual machines, the following is required:

1. The **`libvirtd`** service must be installed and running. This service is provided by the **`libvirt`** package
2. A recent version of the **`mkisofs`** utility must be available. This utility is used to create CD-ROM disk images, and is provided by the **`mkisofs`** package

For Xen installations, the following is also required:

1. A Xen kernel must be running on the executing machine. The running Xen kernel acts as Dom0. In most cases, the virtual machines, called DomUs, will be started under this kernel

   For more information, please refer to the *Red Hat Enterprise Linux 5 Virtualization Guide*
2. The **`pygrub`** program must be available. This program executes virtual machines whose disks contain the kernel they will run. This program is provided by the **`xen`** package.

For KVM installations, the following is also required:

1. A KVM kernel module must be running on the executing machine. The running KVM kernel acts as Dom0. All virtual machines, called DomUs, will be started under this kernel

11.1. Configuring MRG Grid for the virtual machine universe

The configuration files for MRG Grid include various configuration settings for virtual machines. Some settings are required, while others are optional. This section discusses only the required settings.

Initial setup

1. Install the **`condor-vm-gahp`** package:

   ```
   # yum install condor-vm-gahp
   ```

2. Specify the type of virtualization software that is installed, using the **`VM_TYPE`** setting:

   ```
   VM_TYPE = xen
   ```

   Currently, the valid options for the **`VM_TYPE`** setting are:

   * *`xen`*
   * *`kvm`*

3. Specify the location of **`condor_vm-gahp`** and its configuration file, using the **`VM_GAHP_SERVER`** settings:

    ```
    VM_GAHP_SERVER = $(SBIN)/condor_vm-gahp
    ```

4. Set the location for **`condor_vm-gahp`** logs. By default, logs are written to **`/dev/null`**, effectively disabling logging. Change the value of the **`VM_GAHP_LOG`** to enable logging:

    ```
    VM_GAHP_LOG = $(LOG)/VMGahpLogs/VMGahpLog.$(USERNAME)
    ```

5. Set the complete path and filename of the script that controls the VM:

    ```
    VM_SCRIPT = /path/to/filename
    ```

6. Set the maximum amount of memory that the VM is allowed to use (in MB). The value of this parameter depends on how much memory the system administrator will allow VM universe jobs to consume:

    ```
    VM_MEMORY = 512
    ```

7. Set the **`LIBVIRT_XML_SCRIPT`** setting:

    ```
    LIBVIRT_XML_SCRIPT = $(LIBEXEC)/libvirt_simple_script.awk
    ```

8. If networking is required, set the **`VM_NETWORKING`** parameter to *`TRUE`*, and specify the permitted network types using the **`VM_NETWORKING_TYPE`** parameter:

    ```

    VM_NETWORKING = TRUE
    VM_NETWORKING_TYPE = nat, bridge
    ```

    Optionally, the **`VM_NETWORKING_DEFAULT_TYPE`** can also be set. This will allow VM Universe jobs to access the network, even if they have not specified a networking type in their submit description file. To define *`nat`* as the default networking type:

    ```

    VM_NETWORKING_DEFAULT_TYPE = nat
    ```

9. If bridge networking is required, the **`VM_NETWORKING_BRIDGE_INTERFACE`** setting will also need to specified. If it is not defined, then bridge networking will be disabled on the execute node. To specify *`br1`* as the network device:

    ```

    VM_NETWORKING_BRIDGE_INTERFACE = br1
    ```

    The **`VM_NETWORKING_BRIDGE_INTERFACE`** is a string value that must be set to the networking interface that VM Universe jobs (Xen or KVM only) will use for bridge networking. The bridge network interface must be set up by the system administrator prior to setting **`VM_NETWORKING_BRIDGE_INTERFACE`**.

Xen-specific configuration

Additional configuration is necessary for Xen.

1. Although it is not required, it can be necessary to set the default initrd image for Xen to use on Unix-based platforms. Unlike the kernel image, the default initrd image should not be set to the same one used to boot the system. In this case, create a new initrd image by running **`mkinitrd`** from the shell prompt and loading the **`xennet`** and **`xenblk`** drivers into it.

2. Specify the **`XEN_BOOTLOADER`**. The bootloader allows you to select a kernel instead of specifying the Dom0 configuration, and allows the use of the **`xen_kernel = included`** specification when submitting a job to the VM universe. A typical bootloader is **`pygrub`**:

   ```
   XEN_BOOTLOADER = /usr/bin/pygrub
   ```

3. A typical configuration file for Xen is:

   ```
   VM_TYPE = xen
   MAX_VM_GAHP_LOG   = 1000000
   VM_GAHP_DEBUG = D_FULLDEBUG
   VM_MEMORY = 1024
   VM_MAX_MEMORY = 1024
   VM_SCRIPT = $(SBIN)/condor_vm_xen.sh
   XEN_BOOTLOADER = /usr/bin/pygrub
   ```

Restarting MRG Grid with virtualization settings

1. Once the configuration options have been set, restart the **`condor_startd`** daemon on the host. You can do this by running **`condor_restart`**. This should be performed on the central manager machine:

   ```
   $ condor_restart -subsystem startd
   ```

   Note

   If the **`condor_startd`** daemon is currently servicing jobs it will let them finish running before restarting. If you want to force the **`condor_startd`** daemon to restart and kill any running jobs, add the **`-fast`** option to the **`condor_restart`** command.

2. The **`condor_startd`** daemon will pause while it performs the following checks:

   - Exercise the virtual machine capabilities of **`condor_vm-gahp`**
   - Query the properties
   - Advertise the machine to the pool as VM-capable

   If these steps complete successfully, **`condor_status`** will record the virtual machine type and version number. These details can be displayed by running the following command from the shell prompt:

```
$ condor_status -vm machine_name
```

If this command does not display output after some time, it is likely that **condor_vm-gahp** is not able to execute the virtualization software. The problem could be caused by configuration of the virtual machine, the local installation, or a variety of other factors. Check the log file (defined in **VM_GAHP_LOG**) for diagnostics.

3. The VM Universe is only available when MRG Grid is started with the *root* user or administrator. These privileges are required to create a virtual machine on top of a Xen kernel, as well as to use the **libvirtd** utility that controls creation and management of Xen guest virtual machines.

High Availability

MRG Grid can be configured to provide *high availability*. If a machine stops functioning because of scheduled downtime or due to a system failure, other machines can take on key functions. The two key functions that MRG Grid is capable of maintaining are:

- Availability of the job queue - the machine running the **condor_schedd** daemon; and
- Availability of the central manager - the machine running the **condor_negotiator** and **condor_collector** daemons.

This chapter discusses how to set up high availability for both these scenarios.

12.1. High availability of the job queue

The **condor_schedd** daemon controls the job queue. If the job queue is not functioning then the entire pool will be unable to run jobs. This situation can be made worse if one machine is a dedicated submission point for jobs. When a job on the queue is executed, a **condor_shadow** process runs on the machine it was submitted from. The purpose of this process is to handle all input and output fuctionality for the job. However, if the machine running the queue becomes non-functional, **condor_shadow** can not continue communication and no jobs can continue processing.

Without high availability, the job queue would persist, but further jobs would be made to wait until the machine running the **condor_schedd** daemon became available again. By enabling high availability, management of the job queue can be transferred to other designated schedulers and reduce the chance of an outage. If jobs are required to stop without finishing, they can be restarted from the beginning.

To enable high availability, the configuration is adjusted to specify alternate machines that can be used to run the **condor_schedd** daemon. To prevent multiple instances of **condor_schedd** running, a lock is placed on the job queue. When the machine running the job queue fails, the lock is lifted and **condor_schedd** is transferred to another machine. Configuration variables are also used to determine the intervals at which the lock expires, and how frequently polling for expired locks should occur.

When a machine that is able to run the **condor_schedd** daemon is started, the **condor_master** will attempt to discover which machine is currently running the **condor_schedd**. It does this by working out which machine holds a lock. If no lock is currently held, it will assume that no **condor_schedd** is currently running. It will then acquire the lock and start the **condor_schedd** daemon. If a lock is currently held by another machine, the **condor_schedd** daemon will not be started.

The machine running the **condor_schedd** daemon renews the lock periodically. If the machine is not functioning, it will fail to renew the lock, and the lock will become *stale*. The lock can also be released if **condor_off** or **condor_off -schedd** is executed. When another machine that is capable of running **condor_schedd** becomes aware that the lock is stale, it will attempt to acquire the lock and start the **condor_schedd**.

Configuring high availability for the job queue

1. Add the following lines to the local configuration of all machines that are able to run the **condor_schedd** daemon and become 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
```

The **`MASTER_HA_LIST`** macro identifies the **`condor_schedd`** daemon as a daemon that should be kept running.

2. Each machine must have access to the job queue lock. This synchronizes which single machine is currently running the **`condor_schedd`**. **`SPOOL`** identifies the location of the job queue, and needs to be accessible by all High Availability schedulers. This is typically accomplished by placing the **`SPOOL`** directory in a file system that is mounted on all schedulers. **`HA_LOCK_URL`** identifies the location of the job queue lock. Like **`SPOOL`**, this needs to be accessible by all High Availablity Schedulers, and is often found in the same location.

   Always add *`SCHEDD.lock`* to the **`VALID_SPOOL_FILES`** variables. This is to prevent **`condor_preen`** deleting the lock file because it is not aware of it.

Remote job submission

When submitting jobs remotely, the scheduler needs to be identified, using a command such as **`$ condor_submit -remote`** ***`schedd_name myjob`*** **`.submit`**

The command above assumes a single **`condor_schedd`** running on a single machine. When high availability is configured, there are multiple possible **`condor_schedd`** daemons, with any one of them providing a single submission point.

So that jobs can be successfully submitted in a high availability situation, adjust the **`SCHEDD_NAME`** variable in the local configuration of each potential High Availability Scheduler. They will need to have the same value on each machine that could potentially be running the **`condor_schedd`** daemon. Ensure that the value chosen ends with the @ character. This will prevent MRG Grid from modifying the value set for the variable.

```
SCHEDD_NAME = ha-schedd@
```

The command to submit a job is now **`$ condor_submit -remote`** ***`had-schedd@ myjob`*** **`.submit`**

12.2. High availability of the central manager

The **`condor_negotiator`** and **`condor_collector`** daemons are critical to a pool functioning correctly. Both daemons usually run on the same machine, referred to as the *central manager*. If a central manager machine fails, MRG Grid will not be able to match new jobs or allocate new resources. Configuring high availability in a pool reduces the chance of an outage.

High availability allows one of multiple machines within the pool to function as the central manager. While there can be many active **`condor_collector`** daemons, only a single, active **`condor_negotiator`** will be running. The machine with the **`condor_negotiator`** daemon running is the active central manager. All machines running a **`condor_collector`** daemon are idle central managers. All submit and execute machines are configured to report to all potential central manager machines.

Every machine that can potentially be a central manager needs to run the high availability daemon **`condor_had`**. The daemons on each of the machines will communicate to monitor the pool and ensure that a central manager is always available. If the active central manager stops functioning, the **`condor_had`** daemons will detect the failure. The daemons will then select one of the idle machines to become the new active central manager.

If the outage is caused by a network partition, the idle **condor_had** daemons on each side of the partition will choose a new active central manager. As long as the partition exists, there will be an active central manager on each side. When the partition is removed and the network repaired, the **condor_had** daemons will be re-organized and ensure that only one central manager is active.

It is recommended that a single machine is considered the primary central manager. If the primary central manager stops functioning, a secondary central manager can take over. When the primary central manager recovers, it will reclaim central management from the secondary machine. This is particularly useful in situations where the primary central manager is a reliable machine that is expected to have very short periods of instability. An alternative configuration allows the secondary central manager to remain active after the failed central manager machine is restarted.

The high availability mechanism on the central manager operates by monitoring communication between machines. Note that there is a significant difference in communications between machines when:

1. The machine is completely down - crashed or switched off

2. The machine is functioning, but the **condor_had** daemon is not running

The high availability mechanism operates only when the machine is down. If the daemons are simply not running, the system will not select a new active central manager.

The central manager machine records state information, including information about user priorities. Should the primary central manager fail, a pool with high availability enabled would lose this information. Operation would continue, but priorities would be re-initialized. To prevent this occurring, the **condor_replication** daemon replicates the state information on all potential central manager machines. The **condor_replication** daemon needs to be running on the active central manager as well as all potential central managers.

The high availability of central manager machines is enabled through the configuration settings. It is disabled by default. All machines in a pool must be configured appropriately in order to make the high availability mechanism work.

The stabilization period is the time it takes for the **condor_had** daemons to detect a change in the pool state and recover from this change. It is computed using the following formula:

```
stabilization period = 12 * [number of central managers] * $(HAD_CONNECTION_TIMEOUT)
```

Configuring high availability on potential central manager machines

1. Before beginning, remove any parameters from the **NEGOTIATOR_HOST** and **CONDOR_HOST** macros:

   ```
   NEGOTIATOR_HOST=
   CONDOR_HOST=
   ```

2. **Note**

   The following settings must be the same on all potential central manager machines:

   In order to make writing other expressions simpler, define a variable for each potential central manager in the pool.

```
CENTRAL_MANAGER1 = cm1.example.com
CENTRAL_MANAGER2 = cm2.example.com
```

3. List all the potential central managers in the pool:

   ```
   COLLECTOR_HOST = $(CENTRAL_MANAGER1),$(CENTRAL_MANAGER2)
   ```

4. Define a macro for the port number that **condor_had** will listen on. The port number must match the port number used when defining **HAD_LIST**. This port number is arbitrary, but ensure that there are no port number collisions with other applications:

   ```
   HAD_PORT = 51450
   HAD_ARGS = -p $(HAD_PORT)
   ```

5. Define a macro for port number that **condor_replication** will listen on. The port number must match the port number specified for the replication daemon in **REPLICATION_LIST**. The port number is arbitrary, but ensure that there are no port number collisions with other applications:

   ```
   REPLICATION_PORT = 41450
   REPLICATION_ARGS = -p $(REPLICATION_PORT)
   ```

6. Specify a list of addresses for the replication list. It must contain the same addresses as those listed in **HAD_LIST**. Additionally, for each hostname specify the port number of the **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)
   ```

7. Specify a list of addresses for the high availability list. It must contain the same addresses in the same order as the list under **COLLECTOR_HOST**. Additionally, for each hostname specify the port number of the **condor_had** daemon running on that host. The first machine in the list will be considered the primary central manager if **HAD_USE_PRIMARY** is set to *TRUE*:

   ```
   HAD_LIST = $(CENTRAL_MANAGER1):$(HAD_PORT),$(CENTRAL_MANAGER2):$(HAD_PORT)
   ```

8. Specify the high availability daemon connection time. Recommended values are:
   - *2* if the central managers are on the same subnet
   - *5* if security is enabled
   - *10* if the network is very slow, or to reduce the sensitivity of the high availability dameons to network failures

   ```
   HAD_CONNECTION_TIMEOUT = 2
   ```

Important

Setting **`HAD_CONNECTION_TIMEOUT`** value too low can cause the **`condor_had`** daemons to incorrectly assume that the other machines have failed. This can result in a multiple central managers running at once. Conversely, setting the value too high can create a delay in fail-over due to the stabilization period.

The **`HAD_CONNECTION_TIMEOUT`** value is sensitive to the network environment and topology, and should be tuned based on those conditions.

9. Select whether or not to use the first central manager in the **`HAD_LIST`** as a primary central manager:

```
HAD_USE_PRIMARY = true
```

10. Specify which machines have root or administrator privileges within the pool. This is normally set to the machine where the MRG Grid administrator works, provided all users who log in to that machine are trusted:

```
ALLOW_ADMINISTRATOR = $(COLLECTOR_HOST)
```

11. Specify which machines have access to the **`condor_negotiator`**. These are trusted central managers. The default value is appropriate for most pools:

```
ALLOW_NEGOTIATOR = $(COLLECTOR_HOST)
```

12. **Note**

    The following settings can vary between machines. They are master specific parameters:

    Specify the location of executable files:

```
HAD = $(SBIN)/condor_had
REPLICATION = $(SBIN)/condor_replication
```

13. List the daemons that the master central manager should start. It should contain at least the following five daemons:

```
DAEMON_LIST = MASTER, COLLECTOR, NEGOTIATOR, HAD, REPLICATION
DC_DAEMON_LIST = MASTER, COLLECTOR, NEGOTIATOR, HAD, REPLICATION
```

    The **`DC_DAEMON_LIST`** should also include any other daemons running on the node.

14. Specify whether or not to enable the replication feature:

```
HAD_USE_REPLICATION = true
```

15. Name of the file to be replicated:

```
STATE_FILE = $(SPOOL)/Accountantnew.log
```

16. Specify how long (in seconds) to wait in between attempts to replicate the file:

    ```
    REPLICATION_INTERVAL = 300
    ```

17. Specify how long (in seconds) transferer daemons have to complete the download/upload process:

    ```
    MAX_TRANSFERER_LIFETIME = 300
    ```

18. Specify how long (in seconds) for the **condor_had** to wait in between sending ClassAds to the **condor_collector**:

    ```
    HAD_UPDATE_INTERVAL = 300
    ```

19. Specify the master negotiator controllor and the backoff constant:

    ```
    MASTER_NEGOTIATOR_CONTROLLER = HAD
    MASTER_HAD_BACKOFF_CONSTANT = 360
    ```

    Important

    If the backoff constant value is too small, it can result in the **condor_negotiator** churning. This occurs when a constant cycling of the daemons stopping and starting prevents the **condor_negotiator** from being able to run long enough to complete a negotiation cycle. Churning causes an inability for any job to start processing. Increasing the **MASTER_HAD_BACKOFF_CONSTANT** variable can help solve this problem.

20. Specify the maximum size (in bytes) of the log file:

    ```
    MAX_HAD_LOG = 640000
    ```

21. Specify the debug level:

    ```
    HAD_DEBUG = D_COMMAND
    ```

22. Specify the location of the log file for **condor_had**:

    ```
    HAD_LOG = $(LOG)/HADLog
    ```

23. Specify the maximum size (in bytes) of the replication log file:

    ```
    MAX_REPLICATION_LOG = 640000
    ```

24. Specify the debug level for replication:

```
REPLICATION_DEBUG = D_COMMAND
```

25. Specify the location of the log file for **`condor_replication`**:

```
REPLICATION_LOG = $(LOG)/ReplicationLog
```

Configuring high availability on other machines in the pool

Machines that are not potential central managers also require configuration for high availability to work correctly. The following is the procedure for configuring machines that are in the pool, but are not potential central managers.

1. Firstly, remove any parameters from the **`NEGOTIATOR_HOST`** and **`CONDOR_HOST`** macros:

   ```
   NEGOTIATOR_HOST=
   CONDOR_HOST=
   ```

2. Define a variable for each potential central manager:

   ```
   CENTRAL_MANAGER1 = cm1.example.com
   CENTRAL_MANAGER2 = cm2.example.com
   ```

3. Specify a list of all potential central managers:

   ```
   COLLECTOR_HOST = $(CENTRAL_MANAGER1),$(CENTRAL_MANAGER2)
   ```

4. Specify which machines have access to the **`condor_negotiator`**. These are trusted central managers. The default value is appropriate for most pools:

   ```
   ALLOW_NEGOTIATOR = $(COLLECTOR_HOST)
   ```

Using a high availability pool without replication

1. Set the **`HAD_USE_REPLICATION`** configuration variable to *`FALSE`*. This will disable replication at the configuration level.

2. Remove *`REPLICATION`* from both the **`DAEMON_LIST`** and **`DC_DAEMON_LIST`** in the configuration file.

Disabling high availability on the central manager

1. To disable the high availability mechanism on central managers, remove the *`HAD`*, *`REPLICATION`*, and *`NEGOTIATOR`* settings from the **`DAEMON_LIST`** configuration variable on all machines except the primary machine. This will leave only one **`condor_negotiator`** remaining in the pool.

2. To shut down a high availability mechanism that is currently running run the following commands from a host with root or administrator privileges on all central managers:

   a. **`condor_off -all -subsystem negotiator`**

   b. **`condor_off -all -subsystem replication`**

c. **`condor_off -all -subsystem had`**

These commands will kill all the currently running **`condor_had`**, **`condor_replication`** and **`condor_negotiator`** daemons.

3. Run the command **`condor_on -subsystem negotiator`** on the machine where the single **`condor_negotiator`** is going to operate.

Concurrency Limits

It is possible to limit the number of jobs that run simultaneously. This can be used to limit job access to software licences, database connections, shares of overall load on a server, or the number of concurrently run jobs by a particular user or group of users. The restriction is imposed through the use of *concurrency limits*.

Concurrency limits are set when a job is submitted, by specifying the **concurrency_limits** parameter in the job submit file. The **concurrency_limits** parameter references a value in the configuration file. A job submit file can also reference more than one limit.

The **condor_negotiator** uses the information in the submit file when attempting to match the job to a resource. Firstly, it checks that the limits have not been reached. It will then store the limits of the job in the matched machine ClassAd.

Configuration variables for concurrency limits are located in the **condor_negotiator** daemon's configuration file. The important configuration variables for concurrency limits are:

`*_LIMIT`
: In this case, the *** is the name of the limit. This variable sets the allowable number of concurrent jobs for jobs that reference this limit in their submit file. Any number of **`*_LIMIT`** variables can be set, as long as they all have different names

`CONCURRENCY_LIMIT_DEFAULT`
: All limits that are not specified with **`*_LIMIT`**, will use the default limit

This example demonstrates the use of the **`*_LIMIT`** and **`CONCURRENCY_LIMIT_DEFAULT`** configuration variables

In the following configuration file, **`Y_LIMIT`** is set to *2* and **`CONCURRENCY_LIMIT_DEFAULT`** to *1*. In this case, any job that includes the line **`concurrency_limits = y`** in the submit file will have a limit of 2. All other jobs that have a limit other than **Y** will be limited to *1*:

```
CONCURRENCY_LIMIT_DEFAULT = 1
Y_LIMIT = 2
```

The **`*_LIMIT`** variable can also be set without the use of **`CONCURRENCY_LIMIT_DEFAULT`**. With the following configuration, any job that includes the line **`concurrency_limits = x`** in the submit file will have a limit of 5. All other jobs that have a limit other than **X** will not be limited:

```
X_LIMIT = 5
```

Example 13.1. Using **`*_LIMIT`** and **`CONCURRENCY_LIMIT_DEFAULT`**

Creating a job submit file with concurrency limits

1. The **concurrency_limits** attribute references the **`*_LIMIT`** variables:

```
universe = vanilla
executable = /bin/sleep
arguments = 28
concurrency_limits = Y, x, z
```

```
queue 1
```

2. When the job has been submitted, **condor_submit** will sort the given concurrency limits and convert them to lowercase:

   ```
   $ condor_submit job
   Submitting job(s).
   1 job(s) submitted to cluster 28.

   $ condor_q -long 28.0 | grep ConcurrencyLimits
   ConcurrencyLimits = "x,y,z"
   ```

3. Concurrency limits can also be adjusted with **condor_config_val**. In this case, three configuration variables need to be set. Set the **ENABLE_RUNTIME_CONFIG** variable to *TRUE*:

   ```
   ENABLE_RUNTIME_CONFIG = TRUE
   ```

   Allow access from a specific machine to the **CONFIG** access level. This allows you to change the limit from that machine:

   ```
   ALLOW_CONFIG = $(CONDOR_HOST)
   ```

   List the configuration variables that can be changed. The following example allows all limits to be changed, and new limits to be created:

   ```
   NEGOTIATOR.SETTABLE_ATTRS_CONFIG = *_LIMIT
   ```

4. Once the configuration is set, change the limits from the shell prompt:

   ```
   $ condor_config_val -negotiator -rset "X_LIMIT = 3"
   ```

5. After the limits have been changed, reconfigure the **condor_negotiator** to pick up the changes:

   ```
   $ condor_reconfig -negotiator
   ```

6. Information about all concurrency limits can be viewed at the shell prompt by using the **condor_userprio** command with the **-l** option:

   ```
   $ condor_userprio -l | grep ConcurrencyLimit
   ConcurrencyLimit.p = 0
   ConcurrencyLimit.q = 2
   ConcurrencyLimit.x = 6
   ConcurrencyLimit.y = 1
   ConcurrencyLimit.z = 0
   ```

This command displays the current number of jobs using each limit. In the example used above, six jobs are using the *X* limit, two are using the *Q* limit, and none are using the *Z* or *P* limits. The limits with zero users are returned because they have been used at some point in the past. If a limit has been configured but never used, it will not appear in the list.

Note

If jobs are currently using the *X* limit, and **`X_LIMIT`** value is changed to a lower number, all of the original jobs will continue to run. However, no more matches will be accepted against the *X* limit until the number of running jobs drops below the new value.

Dynamic slots

Dynamic slots, also referred to as *partitionable startd*, provides the ability for slots to be marked as partitionable. This allows more than one job to occupy a single slot at any one time. Typically, slots have a fixed set of resources, such as associated CPUs, memory and disk space. By partitioning the slot, those resources can be better utilized.

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 it being matched to new jobs. Because it has been broken up into smaller slots, these will show as running jobs directly. These dynamic slots can also be pre-empted in the same way as ordinary slots.

The original, partionable slot and the new smaller, dynamic slots will be identified individually. The original slot will have an attribute stating **`PartitionableSlot=TRUE`** and the dynamic slots will have an attribute stating **`DynamicSlot=TRUE`**. These attributes can be used in a **`START`** expression to create detailed policies.

This example shows how more than one job can be matched to a single slot through dynamic slots.

In this example, Slot1 has the following resources:

- cpu=10
- memory=10240
- disk=BIG

JobA is allocated to the slot. JobA has the following requirements:

- cpu=3
- memory=1024
- disk=10240

The portion of the slot that is being used is referred to as Slot1.1, and the slot now 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, and so on until the entire resources available have been consumed by jobs.

Example 14.1. Matching multiple jobs to a single slot

Enabling dynamic slots

1. Create a file in the local configuration directory, and add the **`SLOT_TYPE_X_PARTITIONABLE`** configuration variable, with the parameter *TRUE*. The *X* refers to the number of the slot being configured:

   ```
   SLOT_TYPE_X_PARTITIONABLE = TRUE
   ```

2. Save the file

3. Restart the **condor** service:

   ```
   # service condor restart
   Stopping condor services:                [  OK  ]
   Starting condor services:                [  OK  ]
   ```

Submitting jobs to a dynamic pool

In a pool that uses dynamic slots, jobs can have extra desirable resources specified in their submit files:

`request_cpus`
: Defaults to *`1`*

`request_memory`
: Defined in megabytes

  Defaults to the *`ImageSize`* or *`JobVMemory`* parameters.

`request_disk`
: Defined in kilobytes

  Defaults to the *`DiskUsage`* parameter.

This example shows a truncated job submit file, with the requested resources:

```
JobA:
universe = vanilla
executable = ...
...
request_cpus = 3
request_memory = 1024
request_disk = 10240
...
queue
```

Example 14.2. Submitting a job to a dynamic pool

Event Trigger

The *event trigger* service is used to provide diagnostic information about a MRG Grid pool. It uses MRG Messaging to send event notifications about the state of the MRG Grid pool. In most cases, a single event trigger daemon can manage all events for an entire MRG Grid pool.

Each trigger has text associated with it that is used to generate an event if the query conditions are met. The event text can be a simple string, or a complex string that includes ClassAd attributes. ClassAd attributes are specified using **`$(`*`attributedname`*`)`** syntax. For example, if a machine named *`claimedidle`* has been idle for ten minutes and met the **`Idle for long time`** trigger, the following syntax:

```
$(Machine) has been Claimed/Idle for $(TriggerdActivityTime) seconds
```

would create an event with the text:

```
claimedidle has been Claimed/Idle for 600 seconds
```

Configuring the event trigger daemon

1. Create a file in the local configuration directory on all execute nodes, and add the following lines:

   ```
   STARTD_CRON_NAME = TRIGGER_DATA
   STARTD_CRON_AUTOPUBLISH = If_Changed
   TRIGGER_DATA_JOBLIST = GetData
   TRIGGER_DATA_GETDATA_PREFIX = Triggerd
   TRIGGER_DATA_GETDATA_EXECUTABLE = $(BIN)/get_trigger_data
   TRIGGER_DATA_GETDATA_PERIOD = 5m
   TRIGGER_DATA_GETDATA_RECONFIG = FALSE
   ```

2. Create a file in the local configuration directory, and add the following line:

   ```
   DAEMON_LIST = $(DAEMON_LIST), TRIGGERD
   ```

   In this configuration file, other parameters can also be set:

   `TRIGGERD_DEFAULT_EVAL_PERIOD`
   : This controls the default trigger evaluation interval. If not specified, it defaults to evaluating every ten seconds.

   `DATA`
   : This sets the location for the trigger service to save the configured triggers. If not specified, it defaults to the same directory as **`$(SPOOL)`**.

3. Set the IP address and port of the AMQP broker in a file in the local configuration directory on the host that will be running the trigger daemon:

   ```
   QMF_BROKER_HOST = ip/hostname_of_broker
   ```

```
QMF_BROKER_PORT = broker_listen_port
```

If not defined, the broker port will default to 5672.

Initializing Triggers

1. Once the trigger daemon is configured, start MRG Grid. The first time the event trigger service is run, it needs to be initialized with the default set of triggers, using the **condor_trigger_config** tool:

   ```
   $ /usr/sbin/condor_trigger_config -i broker
   ```

   The *broker* parameter should be the name of the broker that communicates with the trigger service.

2. The list of triggers added by the **condor_trigger_config** command are:

   ```
   Trigger Name:                ClassAd Query:
   High CPU Usage               (TriggerdLoadAvg1Min > 5)
   Low Free Mem                 (TriggerdMemFree <= 10240)
   Low Free Disk Space (/)      (TriggerdFilesystem_slash_Free < 10240)
   Busy and Swapping            (State == \"Claimed\" && Activity == \"Busy\" &&
   TriggerdSwapInKBSec > 1000 && TriggerdActivityTime > 300)
   Busy but Idle                (State == \"Claimed\" && Activity == \"Busy\" &&
   CondorLoadAvg < 0.3 && TriggerdActivityTime > 300)
   Idle for long time           (State == \"Claimed\" && Activity == \"Idle\" &&
   TriggerdActivityTime > 300)
   dprintf Logs                 (TriggerdCondorLogDPrintfs != \"\")
   Core Files                   (TriggerdCondorCoreFiles != \"\")
   Logs with ERROR entries     (TriggerdCondorLogCapitalError != \"\")
   Logs with error entries     (TriggerdCondorLogLowerError != \"\")
   Logs with DENIED entries    (TriggerdCondorLogCapitalDenied != \"\")
   Logs with denied entries    (TriggerdCondorLogLowerDenied != \"\")
   Logs with WARNING entries   (TriggerdCondorLogCapitalWarning != \"\")
   Logs with warning entries   (TriggerdCondorLogLowerWarning != \"\")
   Logs with stack dumps       (TriggerdCondorLogStackDump != \"\")
   ```

Adding and Removing Triggers

1. To add a trigger to the service, use the **condor_trigger_config** command with the **-a** option. Specify the name, query, and trigger text, in the following syntax:

   ```
   $ condor_trigger_config -a -n name -q query -t text broker
   ```

   In the above syntax, replace *name* with the name of the trigger, replace *query* with the ClassAd query (which must evaluate to *TRUE* for the trigger to run), and *text* with the string to be raised in the event. The *broker* parameter should be the name of the broker that communicates with the trigger service.

2. To list all triggers that are currently configured, use the **condor_trigger_config** command with the **-l** option, in the following syntax:

```
$ condor_trigger_config -l broker
```

The *broker* parameter should be the name of the broker that communicates with the trigger service.

3. To remove a trigger from the service, use the **condor_trigger_config** command with the **-d** option. Specify the ID number of the trigger, in the following syntax:

```

$ condor_trigger_config -d ID broker
```

   In the above syntax, replace *ID* with the unique ID number of the trigger. The *broker* parameter should be the name of the broker that communicates with the trigger service.

Scheduling to Amazon EC2

The *elastic compute cloud* (EC2) is a service provided by Amazon Web Services. It provides flexible processing power that can be used as an extension to an existing MRG Grid pool in the form of a cloud computing environment.

In addition to the computing power of EC2, Amazon also provides storage, referred to as the *simple storage service* (S3), and a simple queue service (SQS) that provides distributed message queuing capabilities. MRG Grid applications use SQS, are stored in S3, and run in EC2.

In EC2, the cloud resource is referred to as an *Amazon Machine Image* (AMI). EC2 resources are started, monitored, and cleaned up locally. The application is installed in an AMI stored in S3. Once started, the application is responsible for the life-cycle of the job and the termination of the AMI instance.

AMI instances running in EC2 do not have persistent storage directly available. It is advisable to program the AMI to transfer the output from a job out of the running instance before it is shut down.

MRG Grid uses EC2 in two different ways:

- MRG/EC2 Basic
- MRG/EC2 Enhanced

MRG/EC2 Basic uses Amazon EC2 AMIs to perform jobs. AMIs are built to perform a specific job, handle the input and output, and are responsible for shutting down when the job has completed. MRG Grid starts and monitors the AMI during the lifetime of the job.

The MRG/EC2 Enhanced feature is an extension of MRG/EC2 Basic that allows vanilla universe jobs to be run in Amazon's EC2 service. MRG/EC2 Enhanced uses generic AMIs to execute vanilla universe jobs. Jobs executed with MRG/EC2 Enhanced act like any other vanilla universe job, except the execution node is in EC2 instead of a local condor pool.

This chapter contains information on getting and setting up the EC2 Execute Node. It then goes on to provide information on using MRG/EC2 Basic and MRG/EC2 Enhanced. It assumes that you have already got an account with Amazon. For more information on obtaining an Amazon web services (AWS) account, and for Amazon-specific information on EC2, including billing rates, and terms and conditions, visit the *Amazon Web Services website*[1].

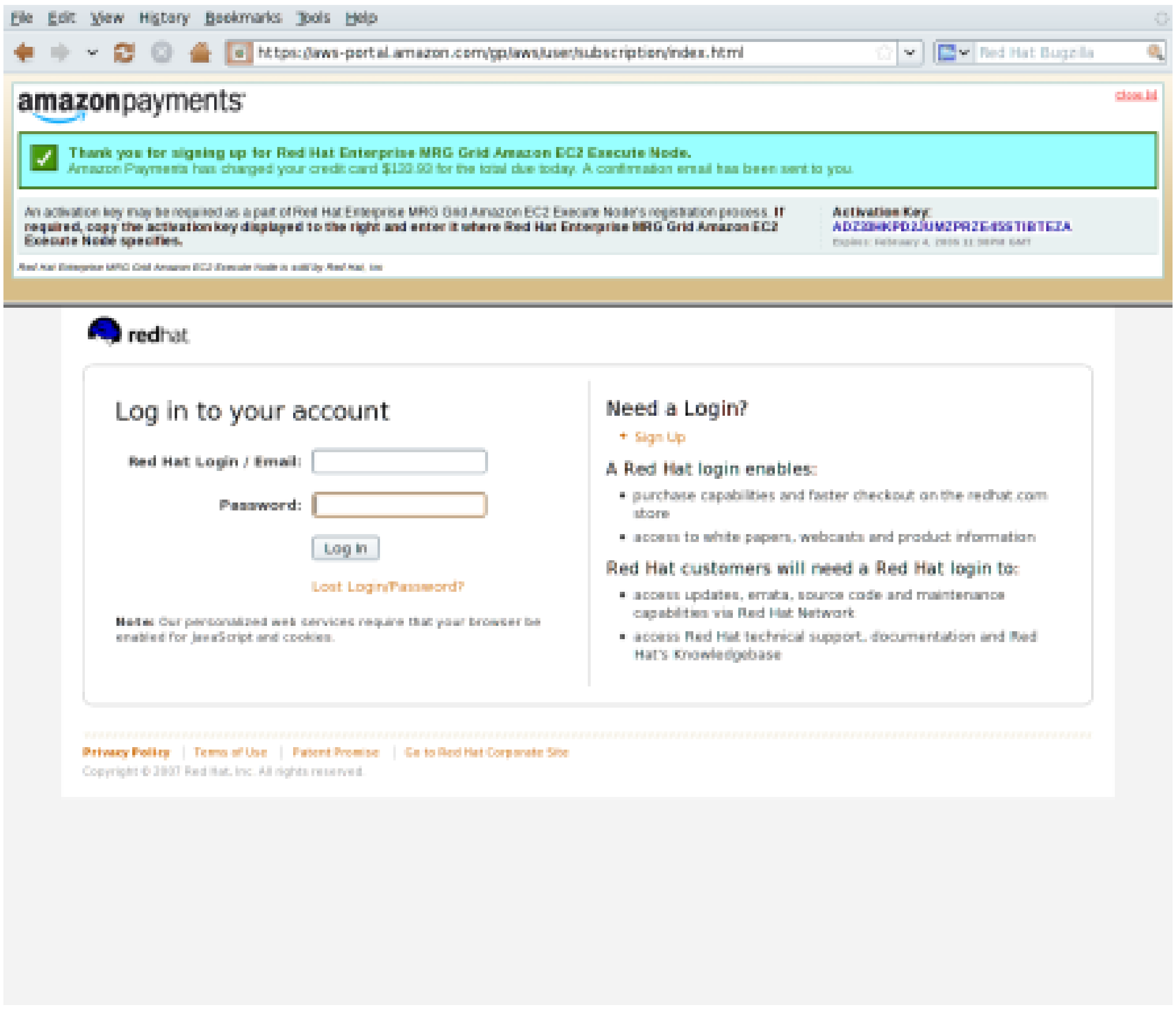
16.1. Getting the MRG Grid Amazon EC2 Execute Node

The *Red Hat Enterprise MRG Grid Amazon EC2 Execute Node* products must be purchased from Amazon. Hourly pricing and additional information can be found at *http://www.redhat.com/solutions/cloud/*

1. Visit the *Amazon product page*[2] and purchase Red Hat Enterprise MRG Grid Amazon EC2 Execute node from Amazon's DevPay service. Enter the purchase information and click on **Place your order**.

2. Once payment has been completed successfully, follow the prompts to log in to the Red Hat Network (RHN).

[1] http://aws.amazon.com/



https://aws-portal.amazon.com/gp/aws/user/subscription/index.html

Thank you for signing up for Red Hat Enterprise MRG Grid Amazon EC2 Execute Node.
Amazon Payments has charged your credit card $120.90 for the total due today. A confirmation email has been sent to you.

An activation key may be required as a part of Red Hat Enterprise MRG Grid Amazon EC2 Execute Node's registration process. **If required, copy the activation key displayed to the right and enter it where Red Hat Enterprise MRG Grid Amazon EC2 Execute Node specifies.**

Activation Key:
ADZ20HKPD2JUM2PR2E4S5T1BTEZA
Expires: February 4, 2009 11:59PM GMT

Red Hat Enterprise MRG Grid Amazon EC2 Execute Node is sold by Red Hat, Inc

Log in to your account

Red Hat Login / Email:

Password:

Lost Login/Password?

Note: Our personalized web services require that your browser be enabled for JavaScript and cookies.

Need a Login?

- Sign Up

A Red Hat login enables:

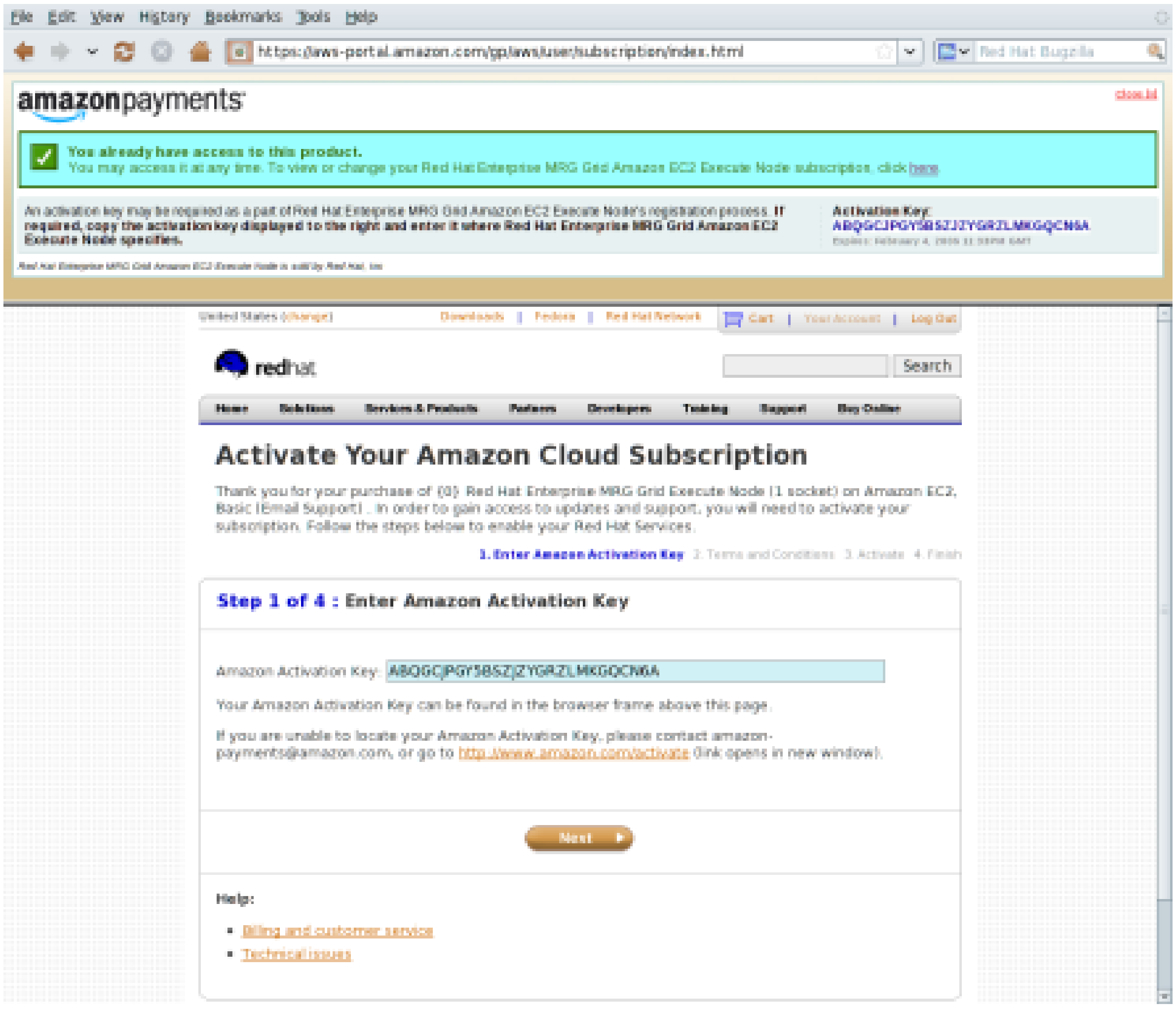
- purchase capabilities and faster checkout on the redhat.com store
- access to white papers, webcasts and product information

Red Hat customers will need a Red Hat login to:

- access updates, errata, source code and maintenance capabilities via Red Hat Network
- access Red Hat technical support, documentation and Red Hat's Knowledgebase

Privacy Policy | Terms of Use | Patent Promise | Go to Red Hat Corporate Site
Copyright © 2007 Red Hat, Inc. All rights reserved.

3. Activate the Amazon Cloud Subscription by completing the four steps on the screen. When the activation has been successfully completed, an email will be sent.


File Edit View History Bookmarks Tools Help
https://aws-portal.amazon.com/gp/aws/user/subscription/index.html
Red Hat Bugzilla
amazonpayments
close

You already have access to this product.
You may access it at any time. To view or change your Red Hat Enterprise MRG Grid Amazon EC2 Execute Node subscription, click here

An activation key may be required as a part of Red Hat Enterprise MRG Grid Amazon EC2 Execute Node's registration process. **If required, copy the activation key displayed to the right and enter it where Red Hat Enterprise MRG Grid Amazon EC2 Execute Node specifies.**

Activation Key:
ABQGCJPGY5BSZJZYGRZLMKGQCN6A
Expires: February 4, 2009 11:59PM GMT

Red Hat Enterprise MRG Grid Amazon EC2 Execute Node is sold by Red Hat, Inc

United States (change)
Downloads | Fedora | Red Hat Network
Cart | Your Account | Log Out
redhat
Search
Home Solutions Services & Products Partners Developers Training Support Buy Online

Activate Your Amazon Cloud Subscription

Thank you for your purchase of (0) Red Hat Enterprise MRG Grid Execute Node (1 socket) on Amazon EC2, Basic (Email Support) . In order to gain access to updates and support, you will need to activate your subscription. Follow the steps below to enable your Red Hat Services.

1. Enter Amazon Activation Key 2. Terms and Conditions 3. Activate 4. Finish

Step 1 of 4 : Enter Amazon Activation Key

Amazon Activation Key: ABQGCJPGY5BSZJZYGRZLMKGQCN6A

Your Amazon Activation Key can be found in the browser frame above this page.

If you are unable to locate your Amazon Activation Key, please contact amazon-payments@amazon.com, or go to http://www.amazon.com/activate (link opens in new window).

Help:

- Billing and customer service
- Technical issues

Note

After logging into RHN a page stating **This is an application to activate Amazon Activation Keys** might be displayed. If this occurs, click the **refresh** button in the browser. You should then be presented with the activation page.

4. An EC2 account with Amazon web services (AWS) is required to be able to connect to the new EC2 instance. The account can be set up from *http://aws.amazon.com*. You will also need a copy of the AWS private key and certificate. These can be found in **Access Identifiers** under the **Your Account** menu in AWS.

   The required tools are available as the **Amazon EC2 API Tools** available from *Amazon Web Services*[3].

Note

For help with getting familiar with EC2, read through the *AWS Getting Started Guide*[4]

5. The following environment variables must be set at the shell prompt before starting:

   - **EC2_HOME**
   - **PATH**
   - **JAVA_HOME**
   - **EC2_CERT**
   - **EC2_PRIVATE_KEY**

   ```
   $ export EC2_HOME=ec2-api-tools-1.2-14611
   $ export PATH=ec2-api-tools-1.2-14611/bin:$PATH
   $ export JAVA_HOME=/usr/lib/jvm/jre
   $ export EC2_CERT=cert-LCNPCCNJ4CQIPO6JTQL6ICZGX.pem
   $ export EC2_PRIVATE_KEY=pk-LCNPCCNJ4CQIPO6JTQL6ICZGX.pem
   ```

6. Once the purchase has been completed, one of two possible Amazon Machine Image (AMI) identification numbers will be provided. These are **`ami-49e70020`** for 32-bit instances and **`ami-5de70034`** for 64-bit instances.

   Once the environment and the AMI have been set up, create an SSH keypair. This can be achieved by using the **`ec2-add-keypair`** command at the shell prompt. Save the private key part locally:

   ```
   $ ec2-add-keypair My-MRG-Grid-Key | tail -n +2 | tee My-MRG-Grid-Key.txt
   ```

   The new MRG Grid instance can now be started using the **`ec2-run-instances`** command. In this example, the *`key (-k)`* name is the name given to the SSH keypair.

   ```
   $ ec2-run-instances ami-49e70020 -k My-MRG-Grid-Key
   RESERVATION    r-cab704a3    126065491017    default
   INSTANCE    i-0dcb4264    ami-49e70020            pending    My-MRG-Grid-Key    0
    m1.small    2009-02-04T23:14:05+0000    us-east-1c    aki-41e70028    ari-43e7002a
   ```

   The EC2 instance begins as *`pending`* and waits for a place to run. Use the **`ec2-describe-instances`** command for the status of the instance:

   ```
   $ ec2-describe-instances
   RESERVATION    r-cab704a3    126065491017    default
   INSTANCE    i-0dcb4264    ami-49e70020 ec2-174-129-129-65.compute-1.amazonaws.com
    domU-12-31-39-00-C1-18.compute-1.internal    running    My-MRG-Grid-Key    0 A3EDFA94
    m1.small    2009-02-04T23:14:05+0000    us-east-1c    aki-41e70028 ari-43e7002a
   ```

   Once the instance is running, it will give you a name to which you can connect. In this case, it is **`ec2-174-129-129-65.compute-1.amazonaws.com`**.

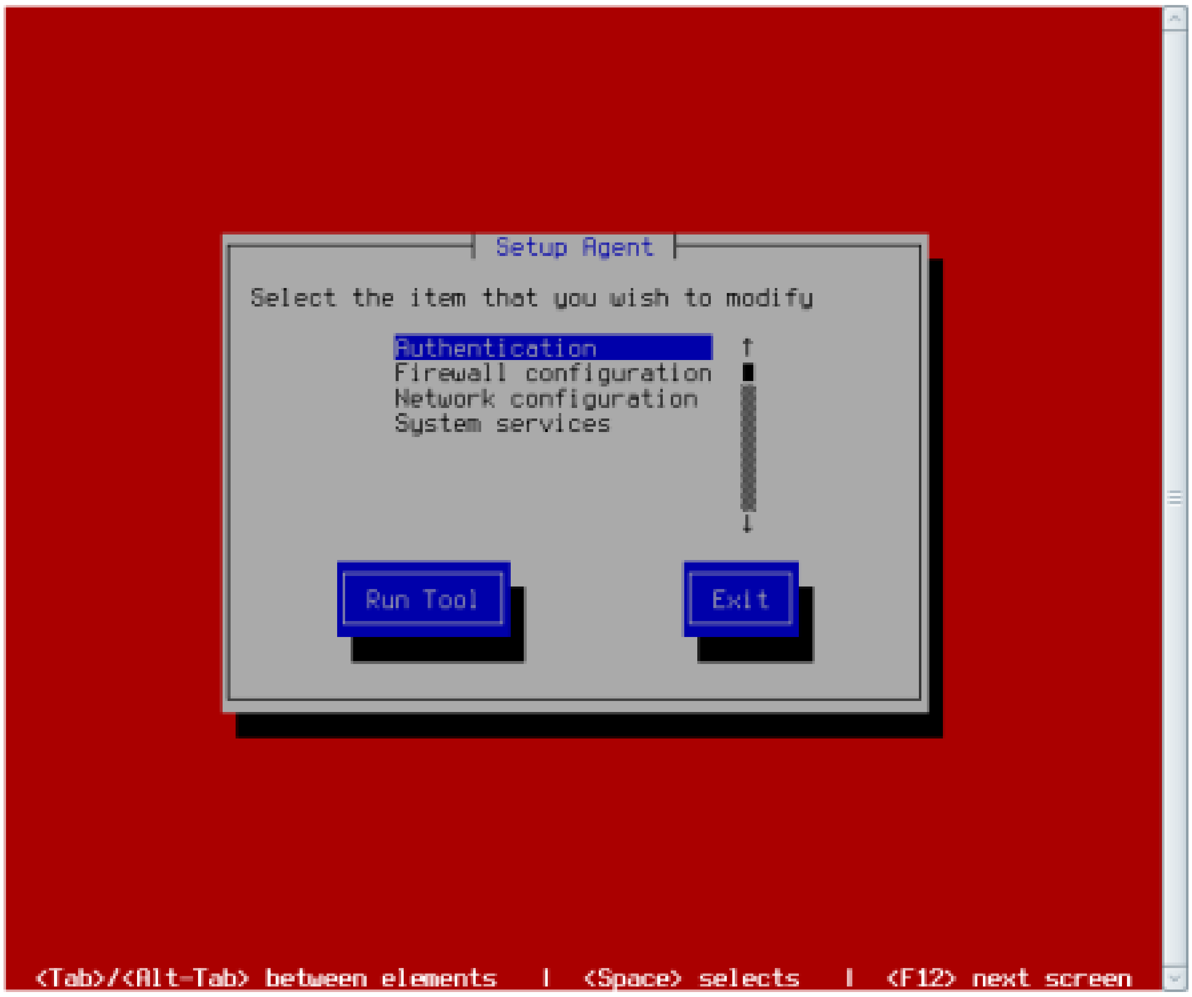
7. Connect to the EC2 instance using **`ssh`**.

Important

The connection to the EC2 instance is performed by the root user.

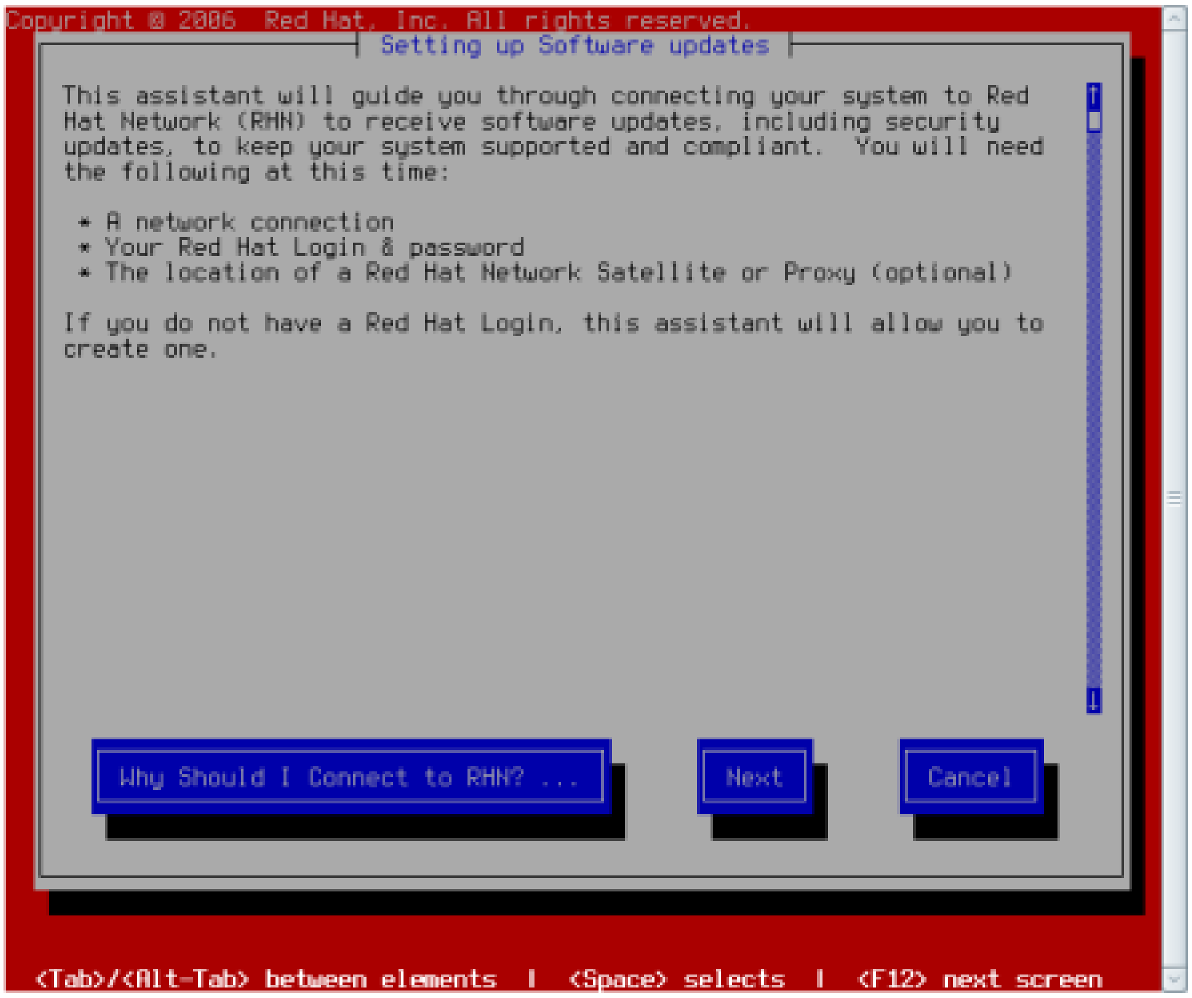
```
$ ssh -i My-MRG-Grid-Key.txt root@ec2-174-129-129-65.compute-1.amazonaws.com
The authenticity of host 'ec2-174-129-129-65.compute-1.amazonaws.com (174.129.129.65)'
 can't be established.
RSA key fingerprint is 71:14:41:cf:75:f3:2a:a2:ee:e8:8e:6e:f7:f7:07:65.
Are you sure you want to continue connecting (yes/no)? yes
Warning: Permanently added
 'ec2-174-129-129-65.compute-1.amazonaws.com,174.129.129.65' (RSA) to the list of known
 hosts.
```

8. Once connected, there is a series of configuration screens. The first one is a **Setup Assistant**. Provide the requested information and select the **Exit** button to finish.



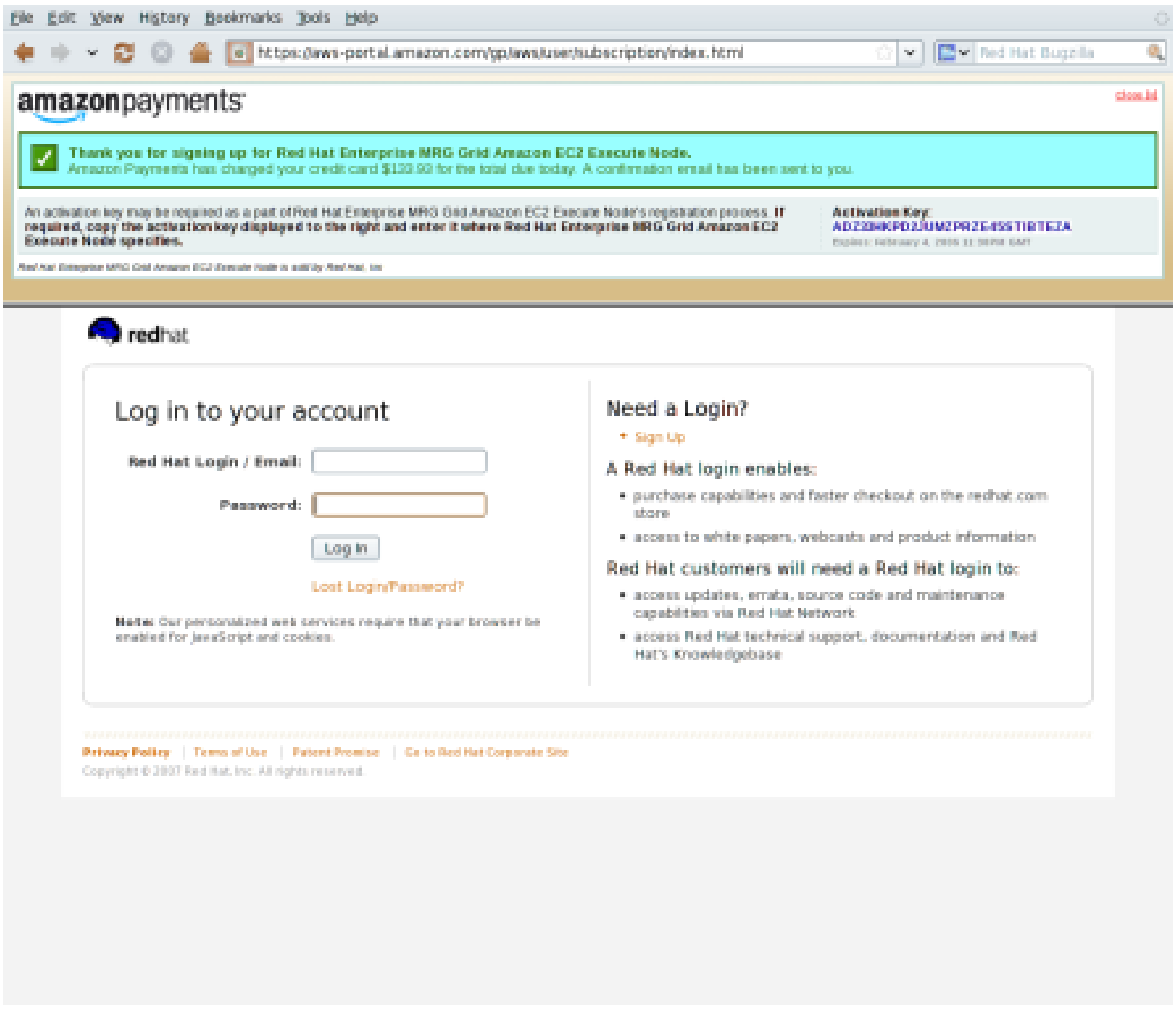
```
Setup Agent

Select the item that you wish to modify

Authentication
Firewall configuration
Network configuration
System services

Run Tool        Exit

<Tab>/<Alt-Tab> between elements   |  <Space> selects   |  <F12> next screen
```

The next screen is **Setting up Software updates**. This screen performs the same function as the **`rhn_register`** command. Select **Next**, enter the RHN login information, and select **Next** to continue through the registration process. The instance must be registered with RHN to get access to the MRG Grid channels.



```
Copyright © 2006  Red Hat, Inc. All rights reserved.
                 Setting up Software updates

  This assistant will guide you through connecting your system to Red
  Hat Network (RHN) to receive software updates, including security
  updates, to keep your system supported and compliant.  You will need
  the following at this time:

   * A network connection
   * Your Red Hat Login & password
   * The location of a Red Hat Network Satellite or Proxy (optional)

  If you do not have a Red Hat Login, this assistant will allow you to
  create one.

     Why Should I Connect to RHN? ...       Next         Cancel

  <Tab>/<Alt-Tab> between elements  |  <Space> selects  |  <F12> next screen
```

9. Once the registration is completed, a root user prompt will be displayed on the EC2 instance. Before the MRG Grid packages can be installed, the MRG Grid channels will need to be enabled through RHN. This can be done by logging in at *http://rhn.redhat.com*.


File Edit View History Bookmarks Tools Help
https://aws-portal.amazon.com/gp/aws/user/subscription/index.html
Red Hat Bugzilla
amazonpayments
close [x]

Thank you for signing up for Red Hat Enterprise MRG Grid Amazon EC2 Execute Node.
Amazon Payments has charged your credit card $120.00 for the total due today. A confirmation email has been sent to you.

An activation key may be required as a part of Red Hat Enterprise MRG Grid Amazon EC2 Execute Node's registration process. **If required, copy the activation key displayed to the right and enter it where Red Hat Enterprise MRG Grid Amazon EC2 Execute Node specifies.**

Activation Key:
ADZ20HKPD2JUM2PRZE4SSTIBTEZA
Expires: February 4, 2009 11:59PM GMT

Red Hat Enterprise MRG Grid Amazon EC2 Execute Node is sold by Red Hat, Inc

Log in to your account

Red Hat Login / Email:

Password:

Note: Our personalized web services require that your browser be enabled for JavaScript and cookies.

Need a Login?

- Sign Up

A Red Hat login enables:

- purchase capabilities and faster checkout on the redhat.com store
- access to white papers, webcasts and product information

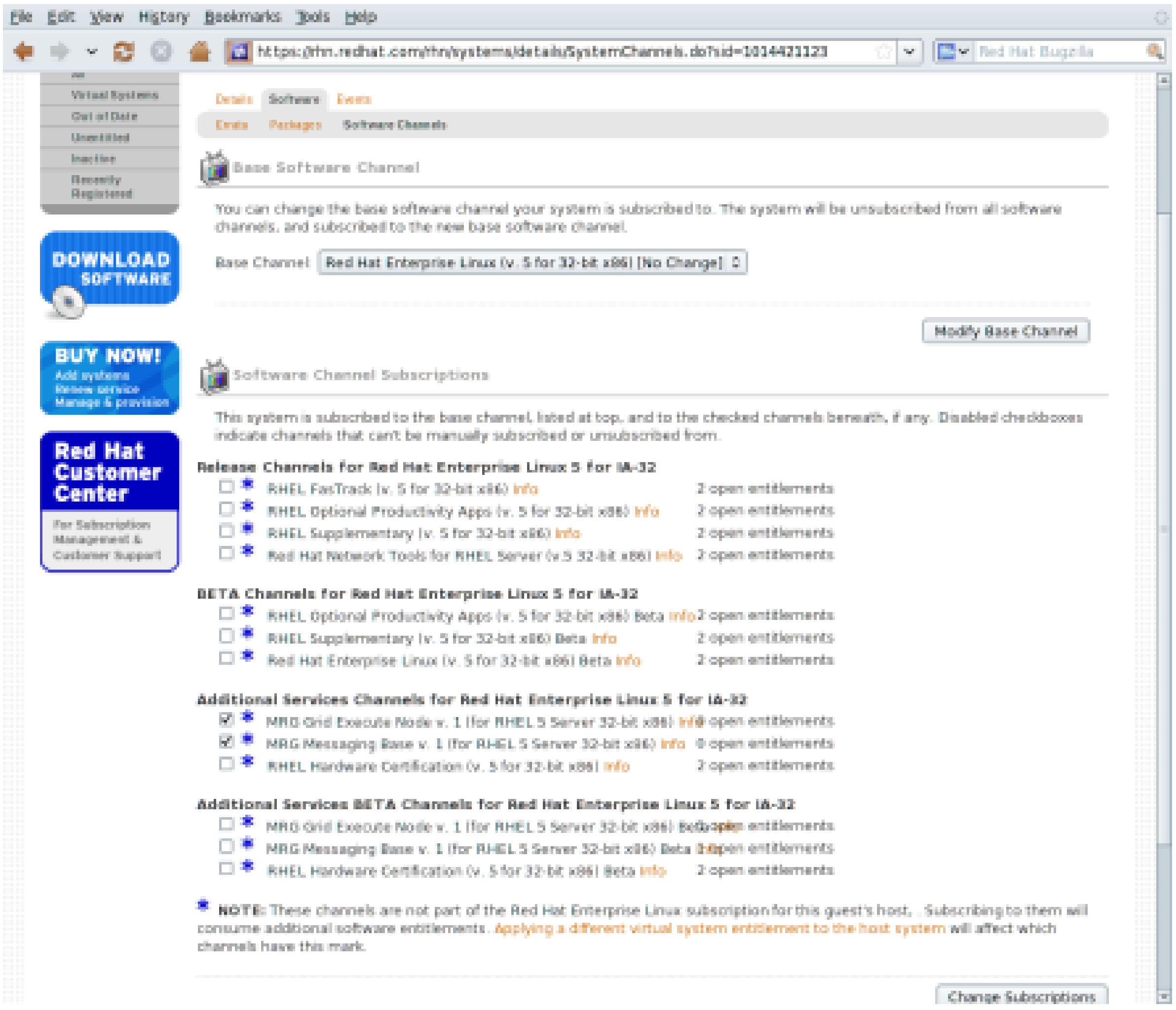
Red Hat customers will need a Red Hat login to:

- access updates, errata, source code and maintenance capabilities via Red Hat Network
- access Red Hat technical support, documentation and Red Hat's Knowledgebase

Privacy Policy | Terms of Use | Patent Promise | Go to Red Hat Corporate Site

Copyright © 2007 Red Hat, Inc. All rights reserved.

Find the registered system and click on it to show the details. Select **Alter Channel Subscriptions**. Under the **Software Channel Subscriptions** menu, select **MRG Grid Execute Node** and **MRG Messaging Base** channels. Save the settings by clicking on the **Change Subscriptions** button.


File Edit View History Bookmarks Tools Help
https://rhn.redhat.com/rhn/systems/details/SystemChannels.do?sid=1014421123
Red Hat Bugzilla
Virtual Systems
Out of Date
Unentitled
Inactive
Recently Registered
DOWNLOAD SOFTWARE
BUY NOW! Add systems Renew service Manage & provision
Red Hat Customer Center
For Subscription Management & Customer Support
Details
Software
Events
Errata
Packages
Software Channels
Modify Base Channel
Change Subscriptions

Base Software Channel

You can change the base software channel your system is subscribed to. The system will be unsubscribed from all software channels, and subscribed to the new base software channel.

Base Channel: Red Hat Enterprise Linux (v. 5 for 32-bit x86) [No Change]

Software Channel Subscriptions

This system is subscribed to the base channel, listed at top, and to the checked channels beneath, if any. Disabled checkboxes indicate channels that can't be manually subscribed or unsubscribed from.

Release Channels for Red Hat Enterprise Linux 5 for IA-32

☐	RHEL FasTrack (v. 5 for 32-bit x86) Info	2 open entitlements
☐	RHEL Optional Productivity Apps (v. 5 for 32-bit x86) Info	2 open entitlements
☐	RHEL Supplementary (v. 5 for 32-bit x86) Info	2 open entitlements
☐	Red Hat Network Tools for RHEL Server (v.5 32-bit x86) Info	2 open entitlements

BETA Channels for Red Hat Enterprise Linux 5 for IA-32

☐	RHEL Optional Productivity Apps (v. 5 for 32-bit x86) Beta Info	2 open entitlements
☐	RHEL Supplementary (v. 5 for 32-bit x86) Beta Info	2 open entitlements
☐	Red Hat Enterprise Linux (v. 5 for 32-bit x86) Beta Info	2 open entitlements

Additional Services Channels for Red Hat Enterprise Linux 5 for IA-32

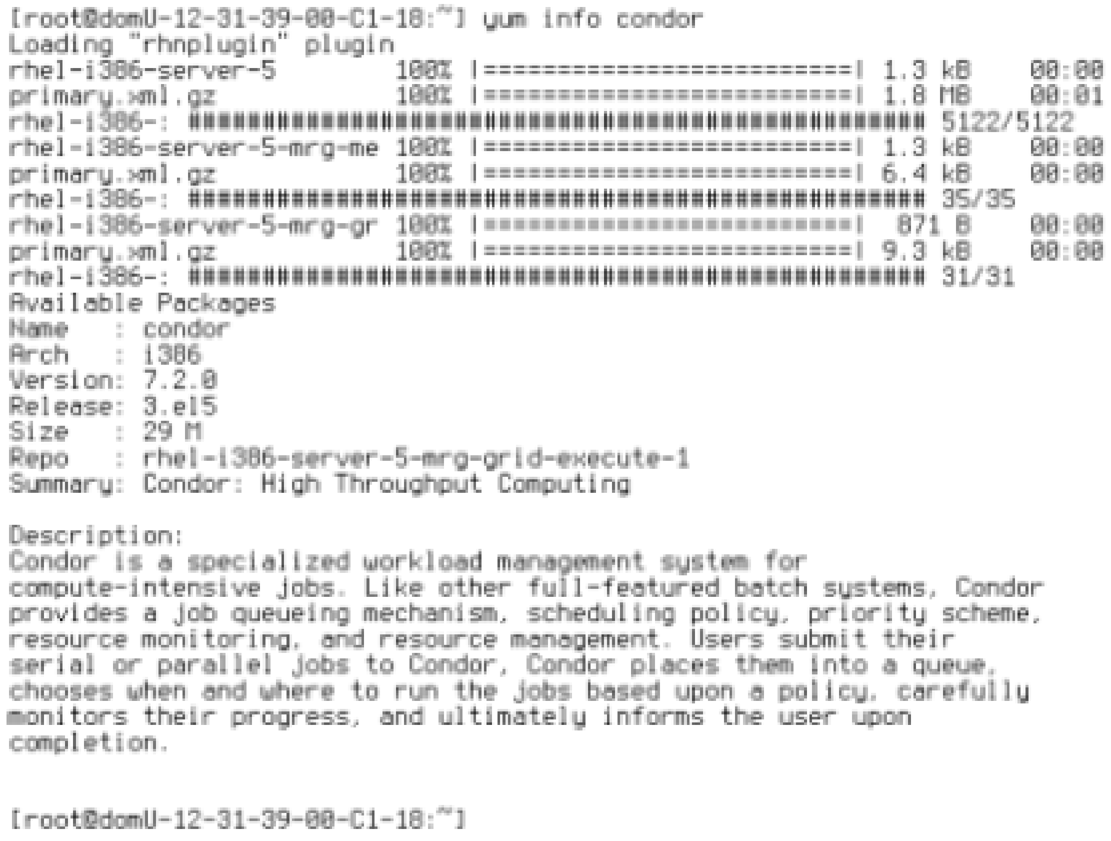
☑	MRG Grid Execute Node v. 1 (for RHEL 5 Server 32-bit x86) Info	0 open entitlements
☑	MRG Messaging Base v. 1 (for RHEL 5 Server 32-bit x86) Info	0 open entitlements
☐	RHEL Hardware Certification (v. 5 for 32-bit x86) Info	2 open entitlements

Additional Services BETA Channels for Red Hat Enterprise Linux 5 for IA-32

☐	MRG Grid Execute Node v. 1 (for RHEL 5 Server 32-bit x86) Beta Info	2 open entitlements
☐	MRG Messaging Base v. 1 (for RHEL 5 Server 32-bit x86) Beta Info	2 open entitlements
☐	RHEL Hardware Certification (v. 5 for 32-bit x86) Beta Info	2 open entitlements

NOTE: These channels are not part of the Red Hat Enterprise Linux subscription for this guest's host. . Subscribing to them will consume additional software entitlements. Applying a different virtual system entitlement to the host system will affect which channels have this mark.

10. Run the **`yum info condor`** command at the shell prompt to verify that you now have access to the MRG Grid channels.



```
[root@domU-12-31-39-00-C1-18:~] yum info condor
Loading "rhnplugin" plugin
rhel-i386-server-5        100% |=========================| 1.3 kB    00:00
primary.xml.gz            100% |=========================| 1.8 MB    00:01
rhel-i386-: ################################################## 5122/5122
rhel-i386-server-5-mrg-me 100% |=========================| 1.3 kB    00:00
primary.xml.gz            100% |=========================| 6.4 kB    00:00
rhel-i386-: ################################################## 35/35
rhel-i386-server-5-mrg-gr 100% |=========================|  871 B    00:00
primary.xml.gz            100% |=========================| 9.3 kB    00:00
rhel-i386-: ################################################## 31/31
Available Packages
Name   : condor
Arch   : i386
Version: 7.2.0
Release: 3.el5
Size   : 29 M
Repo   : rhel-i386-server-5-mrg-grid-execute-1
Summary: Condor: High Throughput Computing

Description:
Condor is a specialized workload management system for
compute-intensive jobs. Like other full-featured batch systems, Condor
provides a job queueing mechanism, scheduling policy, priority scheme,
resource monitoring, and resource management. Users submit their
serial or parallel jobs to Condor, Condor places them into a queue,
chooses when and where to run the jobs based upon a policy, carefully
monitors their progress, and ultimately informs the user upon
completion.


[root@domU-12-31-39-00-C1-18:~]
```

The instance can now be customized. Once this is completed, follow the instructions in the *Amazon Web Services Developer Guide*[5] to rebundle and save the customized API.

16.2. MRG/EC2 Basic

With MRG/EC2 Basic an AMI can be submitted as a job to EC2. This is useful when deploying a complete application stack into EC2. The AMI contains the operating system and all the required packages. EC2 will boot the image and the image can initialize the application stack on boot. MRG/EC2 Basic knowledge is also important when using MRG/EC2 Enhanced.

When setting up MRG Grid for use with EC2 for the first time, the following steps are important:

1. Make changes to your local condor configuration file
2. Prepare the job submission file for EC2 use
3. Set up a security group on EC2 (this step is optional)
4. Submit the job
5. Check that the job is running in EC2
6. Check the image using **ssh** (this step is optional)

Submitting jobs to MRG/EC2 Basic

1. MRG Grid is configured to work with EC2 by default. The necessary configuration settings are in the global configuration file. There is one additional setting you may wish to add to the local configuration:

   ```
   GRIDMANAGER_MAX_SUBMITTED_JOBS_PER_RESOURCE_AMAZON = 10
   ```

   This setting will limit the number of EC2 jobs that can be submitted at any time. AWS has an upper limit of 20. Setting the maximum to less than 20 can help avoid problems.

2. The following is an example of a simple job submission file for MRG/EC2 Basic:

   ```
   # Note to submit an AMI as a job we need the grid universe
   Universe = grid
   grid_resource = amazon

   # Executable in this context is just a label for the job
   Executable  = my_amazon_ec2_job
   transfer_executable = false

   # Keys provided by AWS
   amazon_public_key = cert-ABCDEFGHIJKLMNOPQRSTUVWXYZ.pem
   amazon_private_key = pk-AABBCCDDEEFFGGHHIIJJKKLLMMNNOOPP.pem

   # The AMI ID
   amazon_ami_id = ami-123a456b
   amazon_user_data = Hello EC2!

   # The keypair file if needed for using ssh etc
   amazon_keypair_file = /tmp/keypair

   # The security group for the job
   amazon_security_groups = MY_SEC_GRP

   queue
   ```

   MRG/EC2 Basic requires the grid universe and the *amazon* grid resource. The executable is a label that will show up in the job details when using commands such as **condor_q**. It is not an executable file.

   The AMI ID of the image needs to be specified in the job submission file. User data can also be passed to the remote job if it is required. Applications that require user data can access it using a Representational State Transfer (REST) based interface. Information on how to access image instance data, including user data, is available from *Amazon Web Services Developer Guide*[6].

3. EC2 will provide a keypair for access to the image if required. The **amazon_keypair_file** command specifies where this will be stored.

   EC2 allows users to specify one or more security groups. Security groups can specify what type of access is available. This can include opening specific ports - e.g. port 22 for **ssh** access.

Advanced options

This step is optional.

EC2 provides several options for instance types:

- *m1.small*: i386 instance with 1 compute unit
- *m1.large*: x86_64 instance with 4 compute unit
- *m1.xlarge*: x86_64 instance with 8 compute units
- *c1.medium*: i386 instance with 5 compute units
- *c1.xlarge*: x86_64 instance with 20 compute units

The default instance type is *m1.small* and assumes an i386 architecture. For example, if the AMI you are deploying is x86_64 then you will need to set the following value in your job submission:

```
amazon_instance_type = m1.large
```

For more information on instance types see the *Amazon EC2 Developer Guide*[7].

Note

You could be using the wrong instance type if you see a message like this in your job ClassAd when you run **condor_q -l**:

```
HoldReason = "The requested instance type's architecture (i386) does not
 match the architecture in the manifest for ami-bda347d4 (x86_64)"
```

4. *This step is optional.*

   If **ssh** or other access is required, EC2 provides APIs and commands to create and modify a security group. Download the AMI command line utilities here from the *Amazon Web Services Developer Site*[8]. Documentation on the APIs and command line utilities are also on the *Amazon Web Services Developer Site*[9].

   To use the command line utilities provided by AWS, you will need to set some environment variables.

   Set **EC2_HOME** to point to the location of the tools. The EC2 tools are normally downloaded in a zip file, using version numbers:

```
export EC2_HOME=/home/myuser/ec2-api-tools-X.Y-ZZZZ
```

   EC2 requires X509 certificates. These can be downloaded from your AWS account and set using the following variables:

```
export EC2_CERT=/home/myuser/keys/cert-MPMCVULQDTBLIBUEPGBVK2LIEV6AN6GB.pem
export EC2_PRIVATE_KEY=/home/myuser/keys/pk-MPMCVULQDTBLIBUEPGBVK2LIEV6AN6GB.pem
```

   The EC2 commands require Java, so **JAVA_HOME** must also be set:

```
export JAVA_HOME=/etc/alternatives/jre_1.5.0
```

Use the following commands from the bin directory to create a security group and allow **ssh** access to the AMI. The following are examples. For more information see the documentation at the *Amazon Web Services Developer Guide*[10]. To create a new group called *MY_SEC_GRP* and a short description:

```
./ec2-add-group MY_SEC_GRP -d "My Security Group"
```

Open port 22 and allow **ssh** access:

```
./ec2-authorize MY_SEC_GRP -p 22
```

5. Submit the job using **condor_submit**, as normal.

6. You can check on the status of EC2 jobs, just as regular MRG Grid jobs, by using the **condor_q** and **condor_q -l** commands. When the image has been successfully loaded in EC2 and the job is running, the **condor_q -l** command will show the address of the AMI using the label *AmazonRemoteVirtualMachineName*:

```
$ condor_q -l
AmazonRemoteVirtualMachineName = "ec2-99-111-222-44.compute-1.amazonaws.com"
```

Note

There are tools available for managing running APIs. One of these is the Mozilla™ Firefox™ plugin *Elasticfox*[11].

7. *This step is optional.*

If you are using **ssh** and have opened the appropriate port, **ssh** can also be used to access the running image with a remote shell. The keypair file specified in the job is required:

```
$ ssh -i /tmp/keypair root@ec2-99-111-222-44.compute-1.amazonaws.com
```

This example contains a script for a job to be executed by an AMI. Edit the **/etc/rc.local** file in the AMI and place this code at the end.

This example reads data from the *user-data* field, creates a a file called **output.txt** and transfers that file out of the AMI before shutting down.

```
-- /etc/rc.local --
#!/bin/sh

USER_DATA=`curl http://169.254.169.254/2007-08-29/user-data`
```

```
ARGUMENTS="${USER_DATA%;*}"
RESULTS_FILE="${USER_DATA#*;}"

mkdir /tmp/output
cd /tmp/output

/bin/echo "$ARGUMENTS" > output.txt

cd /tmp
tar czf "$RESULTS_FILE" /tmp/output

curl --ftp-pasv -u user:password -T "$RESULTS_FILE" ftp://server/output

shutdown -h -P now
```

Example 16.1. Creating a script to run an MRG/EC2 Basic job in an AMI

16.3. MRG/EC2 Enhanced

To use MRG/EC2 Enhanced, you will need an Amazon Web Services (AWS) account with access to the following features:

- EC2
- SQS (Simple Queue Service)
- S3 (Simple Storage Service)

This chapter provides instructions on how to download and install the necessary RPMs and Amazon Machine Images (AMIs) for the use and operation of the MRG Grid MRG/EC2 Enhanced feature.

Configuring an Amazon Machine Image

1. On the AMI, use **`yum`** to install the **`condor-ec2-enhanced`** package:

   ```
   # yum install condor-ec2-enhanced
   ```

2. Create a private key file called *`private key`*:

   ```
   $ openssl genrsa -out private_key 1024
   ```

   Create a public key file called *`public_key`*:

   ```
   $ openssl rsa -in private_key -out public_key -pubout
   ```

   Note

   These keys are generated using openssl, and are not the same as the AWS keys needed elsewhere.

   Once the keys have been created, transfer the public key file to a local directory.

Copy the contents of **`private_key`** into the file **`/root/.ec2/rsa_key`** on the AMI. The private key must match the public key set in **`set_rsapublickey`** for a given route or job.

3. The following changes can be specified in any condor configuration file, however it is recommended that they are added to a file located in the local configuration directory:

   Specify the location of the **`condor_startd`** hooks:

```
EC2ENHANCED_HOOK_FETCH_WORK = $(LIBEXEC)/hooks/hook_fetch_work.py
EC2ENHANCED_HOOK_REPLY_FETCH = $(LIBEXEC)/hooks/hook_reply_fetch.py
```

4. Specify the location of the starter hooks:

```
EC2ENHANCED_JOB_HOOK_PREPARE_JOB = $(LIBEXEC)/hooks/hook_prepare_job.py
EC2ENHANCED_JOB_HOOK_UPDATE_JOB_INFO = $(LIBEXEC)/hooks/hook_update_job_status.py
EC2ENHANCED_JOB_HOOK_JOB_EXIT = $(LIBEXEC)/hooks/hook_job_exit.py
```

5. Specify the job hook keywords:

```
STARTD_JOB_HOOK_KEYWORD = EC2ENHANCED
```

6. Set the delay for fetching work and the update interval:

```
FetchWorkDelay = 10
STARTER_UPDATE_INTERVAL = 30
```

7. The **`caroniad`** daemon is used in the MRG/EC2 Enhanced AMI instance to retrieve and process MRG Grid jobs. In order to do this **`caroniad`** communicates with Condor hooks that may or may not be running on the same machine. The daemon is configured by editing the appropriate file in the local configuration directory. The parameters are further described in *Table 16.1, “**`Caroniad`** configuration settings”*.

   Create a file in the local configuration directory, and add the following lines:

```
EC2E_DAEMON = $(SBIN)/caroniad
EC2E_DAEMON_IP = 127.0.0.1
EC2E_DAEMON_PORT = 10000
EC2E_DAEMON_QUEUED_CONNECTIONS = 5
EC2E_DAEMON_LEASE_TIME = 35
EC2E_DAEMON_LEASE_CHECK_INTERVAL = 30
DAEMON_LIST = $(DAEMON_LIST), EC2E_DAEMON
EC2E_DAEMON_LOG = $(LOG)/CaroniaLog
MAX_EC2E_DAEMON_LOG = 1000000
```

   If **`caroniad`** fails to find the configuration variables in the local configuration directory, it will go on to look in **`/etc/condor/caroniad.conf`** instead.

8. The hooks also need to be configured to communicate with **`caroniad`**. The hooks are configured by editing the appropriate file in the local configuration directory. The parameters are further described in *Table 16.1, “**`Caroniad`** configuration settings”*.

   Create a file in the local configuration directory, and add the following lines:

   ```
   JOB_HOOKS_IP = 127.0.0.1
   JOB_HOOKS_PORT = $(EC2E_DAEMON_PORT)
   JOB_HOOKS_LOG = $(LOG)/JobHooksLog
   MAX_JOB_HOOKS_LOG = 10000000
   ```

   If the job hooks fail to find the configuration variables in the local configuration directory, they will go on to look in **`/etc/condor/job-hooks.conf`** instead.

9. Package the AMI. This step will vary depending on how you are building your AMI. If you have changed an existing AMI you should use the following commands (please see the *Amazon Elastic Compute Cloud User Guide*[12] for more information on how to use these commands):

   On the AMI instance run:

   ```
   $ ec2-bundle-vol

   $ ec2-upload-bundle
   ```

   After uploading the bundle it must be registered. On the local machine, register the bundle using the command:

   ```
   $ ec2-register
   ```

   The registration process will return an AMI ID. This ID will be needed when submitting jobs.

Configuration variable	Data type	Description
`EC2E_DAEMON_IP`	IP address	**`caroniad`** will listen on this IP address. **Note** By default, the hooks and **`caroniad`** will run on the same machine. In this case, the loopback IP address is sufficient.

Configuration variable	Data type	Description
`EC2E_DAEMON_PORT`	Integer	The port **`caroniad`** should listen on
`EC2E_DAEMON_QUEUED_CONNECTIONS`	Integer	The number of allowed outstanding connections
`EC2E_DAEMON_LEASE_TIME`	Integer	The amount of time that a job can run without performing an update. If a job has not performed an update within this time frame, it is assumed that an error has occurred and the job will be released or re-sent. This value must be longer than the value specified for **`STARTER_UPDATE_INTERVAL`**.
`EC2E_DAEMON_LEASE_CHECK_INTERVAL`	Integer	The interval to wait between checks to see if a job has had an error
`JOB_HOOKS_IP`	IP Address	The IP address where **`caroniad`** is listening for connections
`JOB_HOOKS_PORT`	Integer	The port **`caroniad`** is listening to for connections
`EC2E_DAEMON_LOG`	String	The location of the log file for **`caroniad`** to use for logging
`MAX_EC2E_DAEMON_LOG`	Integer	The maximum size of the log file before it will be rotated
`JOB_HOOKS_LOG`	String	The location of the log file for the job hooks to use for logging
`MAX_JOB_HOOKS_LOG`	Integer	The maximum size of the job hooks log before rotating

Table 16.1. **`Caroniad`** configuration settings

Download and install the MRG/EC2 Enhanced RPMs

1. The MRG/EC2 Enhanced RPMs can be downloaded using **`yum`**. You will need to ensure that you are connected to the Red Hat Network.

   Important

   For further information on installing Red Hat Enterprise MRG components, see the *MRG Grid Installation Guide*.

2. On the submit machine, use **`yum`** to install the **`condor-ec2-enhanced-hooks`** package:

```
# yum install condor-ec2-enhanced-hooks
```

Configuring the submit machine

1. In order for the local pool to take advantage of the newly created MRG/EC2 Enhanced image, some changes need to be made to the configuration of a submit node in the pool. A sample configuration file for the submit machine is located at **`/usr/share/doc/condor-ec2-enhanced-hooks-1.1/example/condor_config.example`**. Copy the required parts of this file to the local configuration directory of the submit nodes, and edit the following lines to include the AMI ID you received during the registration process:

```
set_amazonamiid = "ami-123a456b";
```

2. Specify the default settings for all routes, including instructions to remove a routed job if it is held or idle for over six hours:

```
JOB_ROUTER_DEFAULTS = \
 [ \
  MaxIdleJobs = 10; \
  MaxJobs = 200; \
\
  set_PeriodicRemove =  (JobStatus == 5 && \
    HoldReason =!= "Spooling input data files") || \
  (JobStatus == 1 && (CurrentTime - QDate) > 3600*6); \
  set_requirements = true; \
  set_WantAWS = false; \
 ]
```

3. Define each routes for sending jobs. Specify a name, a list of requirements and the amazon details:

Note

Just one route is shown here. The example configuration file at **`/usr/share/doc/ec2-enhanced-hooks-1.1/example/condor_config.example`** goes into further detail.

```
JOB_ROUTER_ENTRIES = \
[ GridResource = "condor localhost $(COLLECTOR_HOST)"; \
Name = "Amazon Small"; \
requirements=target.WantAWS is true && (target.Universe is vanilla || target.Universe is
 5) && (target.WantArch is "INTEL" || target.WantArch is UNDEFINED) && (target.WantCpus
 <= 1 || target.WantCpus is UNDEFINED) && (target.WantMemory < 1.7 || target.WantMemory is
 UNDEFINED) && (target.WantDisk < 160 || target.WantDisk is UNDEFINED); \
set_gridresource = "amazon"; \
set_amazonpublickey = "<path_to_AWS_public key>"; \
set_amazonprivatekey = "<path_to_AWS_private_key>"; \
set_amazonaccesskey = "<path_to_AWS_access_key>"; \
set_amazonsecretkey = "<path_to_AWS_secret_key"; \
set_rsapublickey = "<path_to_RSA_public_key>"; \
set_amazoninstancetype = "m1.small"; \
set_amazons3bucketname = "<S3_bucket_name>"; \
set_amazonamiid = "<EC2_AMI_ID>"; \
set_remote_jobuniverse = 5; \
] \
```

The job router entries are described as follows:

- **`set_amazonpublickey`**: The path to a file containing the AWS X.509 public key
- **`set_amazonprivatekey`**: The path to a file containing the AWS X.509 private key
- **`set_amazonaccesskey`**: The path to a file containing the AWS access key
- **`set_amazonsecretkey`**: The path to a file containing the AWS secret key
- **`set_rsapublickey`**: The path to a file containing an RSA public key. This key should match the private key stored in the AMI
- **`set_amazoninstancetype`**: The Amazon EC2 Instance type for the AMI to use with a route
- **`set_amazons3bucketname`**: The Amazon S3 Bucket name condor will use to transfer data for a job
- **`set_amazonamiid`**: The Amazon EC2 Instance ID to use for the route

4. Add the **`JOB_ROUTER`** to the list of daemons to run:

```
DAEMON_LIST = $(DAEMON_LIST) JOB_ROUTER
```

5. Define the polling period for the job router. It is recommended that this value be set to a low value during testing, and a higher value when running on a large scale. This will ensure tests run faster, but prevent using too much CPU when in production:

```
JOB_ROUTER_POLLING_PERIOD = 10
```

6. Set the maximum number of history rotations:

```
MAX_HISTORY_ROTATIONS = 20
```

7. Configure the job router hooks:

```
JOB_ROUTER_HOOK_KEYWORD = EC2E
EC2E_HOOK_TRANSLATE_JOB = $(LIBEXEC)/hooks/hook_translate.py
EC2E_HOOK_UPDATE_JOB_INFO = $(LIBEXEC)/hooks/hook_retrieve_status.py
EC2E_HOOK_JOB_EXIT = $(LIBEXEC)/hooks/hook_job_finalize.py
EC2E_HOOK_JOB_CLEANUP = $(LIBEXEC)/hooks/hook_cleanup.py
EC2E_ATTRS_TO_COPY = EC2RunAttempts, EC2JobSuccessful
```

8. Restart MRG Grid with the new configuration:

```
$ service condor restart
Stopping condor daemon:     [  OK  ]
Starting condor daemon:     [  OK  ]
```

Submitting a job to MRG/EC2 Enhanced

1. A job that uses MRG/EC2 Enhanced is similar to a usual vanilla universe job. However, some keys need to be added to the job submit file. This submit file will cause the job to be routed to the Amazon Small route using administrator defined credentials:

   ```
   universe = vanilla
   executable = /bin/date
   output = date.out
   log = ulog
   requirements = (WantJR =!= true) && (Arch == "INTEL")
   should_transfer_files = yes
   when_to_transfer_output = on_exit
   transfer_executable = false
   +WantAWS = True
   +WantArch = "INTEL"
   +WantCPUs = 1
   +EC2RunAttempts = 0
   queue
   ```

   Important

   The **`requirements`** attribute for the job must include the string **`WantJR =!= true`**. This ensures that the job is only routed to Amazon Web Services. Typically, you would want the requirements **`requirements`** attribute should be set to match the hardware of the AMI the job will run on. For example, if the submit machine is x86_64 and the requirements are not specified, then the above job will not execute because the Amazon Small AMI type is 32-bit, not 64-bit.

2. The following fields are available for routing the job to the correct AMI. If only **`WantAWS`** is defined, then the job will be routed to the small AMI type by default.

   `WantAWS`
   Must be either *`TRUE`* or *`FALSE`*. Use EC2 for executing the job. Defaults to false

   `WantArch`
   Must be either *`INTEL`* or *`X86_64`*. Designates the architecture desired for the job. Defaults to Intel

   `WantCpus`
   Must be an integer. Designates the number of CPUs desired for the job

   `WantMemory`
   Must be a float. Designates the amount of RAM desired for the job, in gigabytes

   `WantDisk`
   Must be an integer. Designates the amount of disk space desired for the job, in gigabytes

3. User credentials for accessing EC2 can be supplied for the submit machine by the site administrator. If this is not the case, the submit file can be used to supply the required information, by adding the following entries:

   ```
   +AmazonAccessKey = "<path>/access_key"
   ```

```
+AmazonSecretKey = "<path>/secret_access_key"
+AmazonPublicKey = "<path>/cert.pem"
+AmazonPrivateKey = "<path>/pk.pem"
+RSAPublicKey = "<path>/rsa_key.pub"
```

These credentials will only be used if the submit machine does not already have credentials defined in **`condor_config`** for the route that the job will use.

Low-latency scheduling

Low-latency scheduling uses the MRG Messaging component of Red Hat Enterprise MRG to allow jobs to execute immediately, bypassing the standard scheduling process. This means a job can begin sooner, and reduces the latency between job submission and execution. The execute nodes in the pool communicate directly with a MRG Messaging broker, which allows any machine capable of sending messages to the broker to submit jobs to the pool.

Submitting a job using condor-low-latency scheduling is similar to submitting a regular Condor job, with the main difference being that instead of using a file for submission the job's attributes are defined in the application headers field of a MRG Messaging message.

When submitting jobs in messages using this method, it is only possible to submit one job for every message. To submit multiple jobs of the same type, multiple messages - each containing one job - will need to be sent to the broker. Messages must have a **`reply-to`** field set, or the jobs will not run. They must also include a unique message ID. If data needs to be submitted with the job, it will need to be compressed and the archive placed in the body of the message.

It is important that messages do not disappear if the broker fails. To avoid this problem, always set the AMQP queues to be durable. Messages containing jobs should also be durable.

The **`caro`** daemon controls the communication between MRG Messaging and MRG Grid. It will look for parameters in the condor configuration files first. It will then look for its own configuration file at **`/etc/condor/carod.conf`**. This file controls the active broker and other options such as the exchange name, message queue and IP information.

Low-latency scheduling also allows a pool of execute nodes to be divided into different projects or priorities. This is achieved by providing different values for the name of the queue. For example, instead of one single queue named *`grid`*, it could be split into two queues named *`grid_high`* and *`grid_low`*. The client program would then be able to use the appropriate routing key to get to the needed queue. This can also be achieved through the API.

Note

For more information on MRG Messaging and the MRG Messaging broker, see the *MRG Messaging User Guide*

Installing and configuring low-latency on execute nodes

1. **Note**

   Because of the way that MRG Messaging handles job submission, there is no need to configure non-execution nodes. These instructions cover installing and configuring execute nodes only.

   Important

   You will require the MRG Messaging broker from the Red Hat Network in order to use low-latency scheduling. For instructions on downloading and configuring the MRG Messaging packages, see the *MRG Messaging Installation Guide*.

You will require the following packages, in addition to the MRG Messaging components:

- condor-low-latency
- condor-job-hooks
- python-condorutils

Use **yum** to install these components:

```
# yum install condor-low-latency

# yum install condor-job-hooks

# yum install python-condorutils
```

2. Create a file in the local configuration directory, and add the following lines:

```
LL_DAEMON = $(SBIN)/carod
LL_BROKER_IP = <broker ip>
LL_BROKER_PORT = 5672
LL_BROKER_QUEUE = grid
LL_DAEMON_IP = 127.0.0.1
LL_DAEMON_PORT = 10000
LL_DAEMON_QUEUED_CONNECTIONS = 5
LL_DAEMON_LEASE_TIME = 35
LL_DAEMON_LEASE_CHECK_INTERVAL = 30
JOB_HOOKS_IP = 127.0.0.1
JOB_HOOKS_PORT = $(LL_DAEMON_PORT)
DAEMON_LIST = $(DAEMON_LIST), LL_DAEMON

# Startd hooks
LOW_LATENCY_HOOK_FETCH_WORK = $(LIBEXEC)/hooks/hook_fetch_work.py
LOW_LATENCY_HOOK_REPLY_FETCH = $(LIBEXEC)/hooks/hook_reply_fetch.py

# Starter hooks
LOW_LATENCY_JOB_HOOK_PREPARE_JOB = $(LIBEXEC)/hooks/hook_prepare_job.py
LOW_LATENCY_JOB_HOOK_UPDATE_JOB_INFO = \
$(LIBEXEC)/hooks/hook_update_job_status.py
LOW_LATENCY_JOB_HOOK_JOB_EXIT = $(LIBEXEC)/hooks/hook_job_exit.py

STARTD_JOB_HOOK_KEYWORD = LOW_LATENCY

LL_DAEMON_LOG = $(LOG)/CaroLog
MAX_LL_DAEMON_LOG = 1000000

JOB_HOOKS_LOG = $(LOG)/JobHooksLog
MAX_JOB_HOOKS_LOG = 10000000
```

For a description of each of these parameters, see *Table 17.1, “Low latency configuration settings”*.

3. Set the **`FetchWorkDelay`** setting. This setting controls how often the condor-low-latency feature will look for jobs to execute, in seconds:

```
FetchWorkDelay = ifThenElse(State == "Claimed" && Activity == "Idle", 0, 10)
STARTER_UPDATE_INTERVAL = 30
```

4. Start the MRG Messaging broker:

```
# service qpidd start
Starting qpidd daemon:                             [  OK  ]
```

5. Restart the **condor** service:

```
# service condor restart
Stopping condor services:                        [  OK  ]
Starting condor services:                        [  OK  ]
```

6. There are some differences between the job submit files of an ordinary job and a low latency job. To ensure the fields are correct, a normal Condor job submission file can be translated into the appropriate fields for the application headers by using the **condor_submit** command with the **-dump** option:

```
$ condor_submit myjob.submit -dump output_file
```

This command produces a file named ***output_file***. This file contains the information contained in the **myjob.submit** in a format suitable for placing directly into the the application header of a message. This method only works when queuing a single message at a time.

Important

The **myjob.submit** should only have one **queue** command with no arguments. For example:

```
executable = /bin/echo
arguments = "Hello there!"
queue
```

Configuration variable	Data type	Description
LL_BROKER_IP	IP Address	The IP address of the broker that **carod** is talking to
LL_BROKER_PORT	Integer	The port on **$(LL_BROKER_IP)** that the broker is listening to
LL_BROKER_QUEUE	String	The queue on the broker for condor jobs
LL_DAEMON_IP	IP Address	The IP address of the interface **carod** is using for connections

Configuration variable	Data type	Description
`LL_DAEMON_PORT`	Integer	The port **`carod`** is listening to for connections
`LL_DAEMON_QUEUED_CONNECTIONS`	Integer	The number of allowed outstanding connections
`LL_DAEMON_LEASE_TIME`	Integer	The maximum amount of time (in seconds) a job is allowed to run without providing an update
`LL_DAEMON_LEASE_CHECK_INTERVAL`	Integer	How often (in seconds) **`carod`** is checking for lease expiration
`LL_DAEMON_LOG`	String	The location of the file **`carod`** should use for logging
`MAX_LL_DAEMON_LOG`	Integer	The maximum size of the **`carod`** log file before being rotated
`JOB_HOOKS_IP`	IP Address	The IP address where **`carod`** is listening for connections
`JOB_HOOKS_PORT`	Integer	The port **`carod`** is listening to for connections
`JOB_HOOKS_LOG`	String	The location of the log file for the job hooks to use for logging
`MAX_JOB_HOOKS_LOG`	Integer	The maximum size of the job hooks log before rotating

Table 17.1. Low latency configuration settings

This example submits a simple low-latency job, by including the job details into the *`application_headers`* field.

The following code excerpt sends a job that will sleep for 5 seconds, and then send the results to the *`replyTo reply-t`* queue. It also ensures that the AMQP message has a unique ID.

```
work_headers = {}
work_headers['Cmd'] = '"/bin/sleep"'
work_headers['Arguments'] = '"5"'
work_headers['Iwd'] = '"/tmp"'
work_headers['Owner'] = '"nobody"'
work_headers['JobUniverse'] = 5
message_props = session.message_properties(application_headers=work_headers)
replyTo = str(uuid4())
message_props.reply_to = session.reply_to('amq.direct', replyTo)
message_props.message_id = uuid4()
```

Example 17.1. Submitting a low-latency job

DAGMan

MRG Grid allows jobs to be submitted and executed in parallel. However, some large-scale computing applications require individual jobs to be processed as an orderly set of dependencies. This is used in some simulations of financial instrument models, or complex 3D modeling.

A *directed acyclic graph* (DAG) is a method of expressing dependencies between jobs. It is a specification that requires certain tasks to be completed before others. Tasks cannot loop, and there must be always be a deterministic path between tasks.

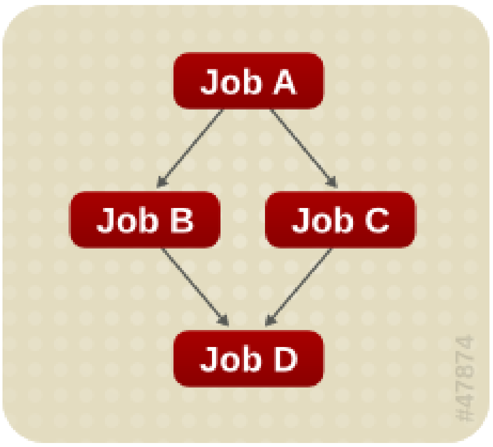
DAGMan (DAG manager) performs workflow management within MRG Grid. Individual jobs are treated as tasks, and each job must be completed before the next can start. The output of a job can be used as the input for the next job.

18.1. DAGMan jobs

DAGMan jobs are submitted to the **condor_schedd** in the same way as ordinary MRG Grid jobs. The **condor_schedd** launches the DAG job inside the scheduler universe.

Submitting and monitoring DAG jobs

DAGMan submit description files act as pointers for the individual job submit description files, and instruct MRG Grid on the order to run the jobs. In the DAGMan submit description file, job dependencies are expressed as PARENT-CHILD relationships. The basic configuration for DAGMan jobs usually resembles a diamond.



In this configuration, Job A must complete successfully. Then Jobs B and C will run concurrently. When Jobs B and C have both completed succesfully, Job D will run.

The submit description file syntax for a simple diamond-shaped DAG is:

```
# this file is called diamond.dag
JOB A A_job.submit
JOB B B_job.submit
JOB C C_job.submit
JOB D D_job.submit
PARENT A CHILD B,C
PARENT B,C CHILD D
```

1. Define the jobs. Each job must have a name, and a location. The location is the submit description file for the individual job.

2. Define each of the parent/child pairs. Parent jobs are the jobs that must be run first. Child jobs cannot be run until the parent jobs have been successfully completed. The pairs must be defined at every level, with each level on a new line. Use commas to specify more than one job.

3. There is no need to specify the scheduler universe in the DAG submit description files. This condition is implied by the **`condor_submit_dag`** tool.

4. When the submit description files are complete and saved to a file, the DAG can be submitted to MRG Grid. This is done using the **`condor_submit_dag`** tool.

   From the shell prompt, use the **`condor_submit_dag`** command with the name of the DAG submit description file:

   ```
   $ condor_submit_dag diamond_dag

   Checking all your submit files for log file names.
   This might take a while...
   Done.
   -----------------------------------------------------------------------
   File for submitting this DAG to Condor           : diamond_dag.condor.sub
   Log of DAGMan debugging messages                 : diamond_dag.dagman.out
   Log of Condor library output                     : diamond_dag.lib.out
   Log of Condor library error messages             : diamond_dag.lib.err
   Log of the life of condor_dagman itself          : diamond_dag.dagman.log

   Submitting job(s).
   Logging submit event(s).
   1 job(s) submitted to cluster 30072.
   -----------------------------------------------------------------------
   ```

   If the job has been submitted successfully, the **`condor_submit_dag`** tool will provide a summary of the submission, including the location of the log files.

   The most important log file to note is the **`condor_dagman`** log labeled *`Log of the life of condor_dagman itself`* and referred to as the *lifetime* log. This file is used to coordinate job execution.

5. The DAG is considered invalid if it has cycles or loops in it. This will be picked up during the DAGMan consistency checking routine, and the DAG will exit with a message reading: **`ERROR: a cycle exists in the DAG`**.

6. There are two methods for submitting more than one DAG. If the list of DAG submit description files are added to the **`condor_submit_dag`** command, all files will be submitted as one large DAG submission:

   ```
   $ condor_submit_dag dag_file1, dag_file2, dag_file3
   ```

   When DAGs are submitted together in this way, there are two important details to note:

   - All the DAGs will execute under a single **`condor_dagman`** process.

- Any nodes that appear multiple times in a single submission will only be executed once.

Alternatively, run the **`condor_submit_dag`** command multiple times, specifying each individual DAG submit description file. In this case, ensure that the DAG submit description files and job names are all unique, to avoid log and output files being overwritten.

7. DAG jobs can be monitored using the **`condor_q`** command. By specifying the username, the results will show only jobs submitted by that user:

```
$ condor_q daguser

29017.0   daguser       6/24 17:22   4+15:12:28 H  0   2.7  condor_dagman
29021.0   daguser       6/24 17:22   4+15:12:27 H  0   2.7  condor_dagman
29030.0   daguser       6/24 17:22   4+15:12:34 H  0   2.7  condor_dagman
30047.0   daguser       6/29 09:13   0+00:01:56 R  0   2.7  condor_dagman
30048.0   daguser       6/29 09:13   0+00:01:07 R  0   2.7  condor_dagman
30049.0   daguser       6/29 09:14   0+00:01:07 R  0   2.7  condor_dagman
30050.0   daguser       6/29 09:14   0+00:01:06 R  0   2.7  condor_dagman
30051.0   daguser       6/29 09:14   0+00:01:06 R  0   2.7  condor_dagman
30054.0   daguser       6/29 09:15   0+00:00:01 R  0   0.0  uname -n
30055.0   daguser       6/29 09:15   0+00:00:00 R  0   0.0  uname -n
30056.0   daguser       6/29 09:15   0+00:00:00 I  0   0.0  uname -n
30057.0   daguser       6/29 09:15   0+00:00:00 I  0   0.0  uname -n
30058.0   daguser       6/29 09:15   0+00:00:00 I  0   0.0  uname -n
30059.0   daguser       6/29 09:15   0+00:00:00 I  0   0.0  uname -n
30060.0   daguser       6/29 09:15   0+00:00:00 I  0   0.0  uname -n

15 jobs; 5 idle, 7 running, 3 held
```

The output of the **`condor_q`** command lists the supervising **`condor_dagman`** jobs.

8. To see extra information about DAG jobs, including how the jobs and processes relate to each other, use the **`condor_q`** command with the **`-dag`** option:

```
$ condor_q -dag daguser

29017.0   daguser       6/24 17:22   4+15:12:28 H  0   2.7  condor_dagman -f -
29021.0   daguser       6/24 17:22   4+15:12:27 H  0   2.7  condor_dagman -f -
29030.0   daguser       6/24 17:22   4+15:12:34 H  0   2.7  condor_dagman -f -
30047.0   daguser       6/29 09:13   0+00:01:50 R  0   2.7  condor_dagman -f -
30057.0    |-B0         6/29 09:15   0+00:00:00 I  0   0.0  uname -n
30058.0    |-C0         6/29 09:15   0+00:00:00 I  0   0.0  uname -n
30048.0   daguser       6/29 09:13   0+00:01:01 R  0   2.7  condor_dagman -f -
30055.0    |-A1         6/29 09:15   0+00:00:00 I  0   0.0  uname -n
30049.0   daguser       6/29 09:14   0+00:01:01 R  0   2.7  condor_dagman -f -
30056.0    |-A2         6/29 09:15   0+00:00:00 I  0   0.0  uname -n
30050.0   daguser       6/29 09:14   0+00:01:00 R  0   2.7  condor_dagman -f -
30059.0    |-B3         6/29 09:15   0+00:00:00 I  0   0.0  uname -n
30060.0    |-C3         6/29 09:15   0+00:00:00 I  0   0.0  uname -n
30051.0   daguser       6/29 09:14   0+00:01:00 R  0   2.7  condor_dagman -f -
30054.0    |-A4         6/29 09:15   0+00:00:00 I  0   0.0  uname -n

15 jobs; 7 idle, 5 running, 3 held
```

Using the **`-dag`** option will show the DAGMan processes that are running, with their associated node listings. In this example, *`Job 30047.0`* is processing child nodes labelled *`B0`* and *`C0`*. These are the names they were given in the DAG submit file. Each DAG manages its own set of nodes, so the names of nodes can be traced back to the DAG submit file for each.

9. To remove a DAG job, use the **`condor_rm`** command with the job number. When a DAG job is removed, all jobs associated with it will also be removed.

   ```
   $ condor_rm 29017.0
   ```

10. When submitting multiple concurrent DAGs to DAGMan, it is useful to ensure that each DAG has its own file area. The easiest way to do this is to ensure that each DAG has a dedicated directory to store its DAG submission, log, output, and error files.

11. The DAG submit tool contains a **`-no_submit`** option. This instructs DAGman to generate a submission file that can be used by DAGman, but does not submit the job to MRG Grid. This is an advanced feature that allows additional editing of the original DAG submit file prior to the submission. It can also be used by external tooling for pre-processing and deferred DAG submissions:

    ```
    $ condor_submit_dag -no_submit diamond_dag
    ```

Declaring variables

Variables can be declared in a DAG submit file, which can then be expanded at runtime. This provides the ability to use a central location to store reusable declarations, such as directory paths, or arguments for jobs.

DAGMan looks for the variable association based on the name of the job. The value of the variable must be declared inside double quotation marks (").

1. In the DAG submit file, include a **VARS** declaration:

   ```
   JOB  A  A_job.submit
   JOB   B  B_job.submit
   JOB   C  C_job.submit
   JOB   D  D_job.submit
   PARENT  A  CHILD B,C
   PARENT  B,C  CHILD D
   VARS  A  dataset="small_data.txt"
   ```

2. The variables are referenced in the individual node submit file:

   ```
   # node job filename: A_job.submit
   executable = A_job
   log        = A.log
   error      = A.err
   arguments =  $(dataset)
   queue
   ```

3. When the node job *A* is executed, it will be launched as:

```
A_job small_data.txt
```

In this example, the argument will only be passed to the **`A_job.submit`** file and not the others at runtime.

Rescuing a DAG

In some cases, a DAG might not be able to completely execute all its node jobs. DAGMan can rescue this work if it has generated a *rescue file* to hold the overall progress of the DAG. This is especially useful for large, complex DAGs that can involve hours of resources.

When a DAG cannot proceed to conclusion due to failures at a node (or nodes), DAGMan will terminate the execution and generate a rescue file. The rescue file will be functionally equivalent to the original DAG. Each job that has successfully completed will be marked by the keyword **`DONE`**. Rescue files have the same filename as the original DAG, with **`.rescue`*`XXX`*** appended, and by default are written to the same directory as the original DAG submit file.

1. Re-submit a rescue DAG by invoking the original DAG submission command:

   ```
   $ condor_submit_dag diamond_dag
   ```

2. DAGMan will check to see if there are any existing rescue DAG files and will use the latest version available. For example, if there is a **`diamond_dag.rescue003`** present and no file with a larger increment, then the DAG will be recovered from that file.

   This behavior can be changed on the **`condor_submit_dag`** command line by using the command:

   ```
   $ condor_submit_dag -DoRescueFrom 2 diamond_dag
   ```

   This will instruct DAG rescue operations to always start from the DAG rescue file number specified. Note that number should be input as *`1`*, *`2`*, or *`3`* and not *`001`*, *`002`*, or *`003`*.

Pre-and-Post script processing

Execution of a DAG can sometimes require the setup and teardown of files and resources external to DAG job executables. In some cases, it might be necessary to copy and decompress zipped data files to a staging location prior to the DAG being run. Once the DAG has executed, those same resources need to be cleaned up. This can be achieved by using the pre- and post-script processing feature of DAGMan.

1. To run pre- and post-scripts around the core DAG job, add **`PRE`** and **`POST`** lines to the DAG submit file:

   ```
   JOB   A    A_job.submit
   JOB    B    B_job.submit
   JOB    C    C_job.submit
   JOB    D    D_job.submit
   PARENT   A    CHILD B,C
   PARENT   B,C    CHILD D
   ```

```
SCRIPT  PRE  C setup_data.sh $JOB
SCRIPT  POST C teardown_data.sh $JOB
```

2. It is also possible to pass a *$JOB* argument in the script, which represents a job name. This can be useful for using the job name as part of an external filename or directory.

3. Create the files referenced in the DAG submit file. The content of **`setup_data.sh`** might be something like this:

   ```
   #!/bin/csh
   tar -C staging/$argv[1] -zxf /mnt/storage/$argv[1]/data.tar.gz
   ```

   The content of **`teardown_data.sh`** might be:

   ```
   #!/bin/csh
   rm -fr staging/$argv[1]
   ```

   This assumes that the node jobs will be set up to use the data from their respective staging directories.

4. DAGMan monitors the return values of the node job itself, and also the scripts. DAGMan judges success or failure in the following way:

   - If a pre-script returns failure, neither the node job nor the post-script will be executed
   - In a node without a post-script, if any one job returns a failure the entire node is considered to have failed.
   - If both the pre-script and post-script in a node return success, the entire node is considered to have succeeded, regardless of the outcome of any one job.

Nesting DAGs

DAG jobs are static representations that have been completely defined prior to submission. These static representations can be composed not only from individual node jobs, but also from other DAG jobs by means of *nesting*. This makes it easier to manage and visualize large scale DAG jobs in a hierarchical way. This makes it possible to focus on developing and testing smaller individual DAGs before integrating them into a larger workflow.

1. In order to nest a DAG inside another DAG as a node, create an inner DAG:

   ```
   # inner.dag
   JOB   X  X_job
   JOB   Y  Y_job
   JOB   Z  Z_job
   PARENT  X  CHILD Y
   PARENT  Y  CHILD Z
   ```

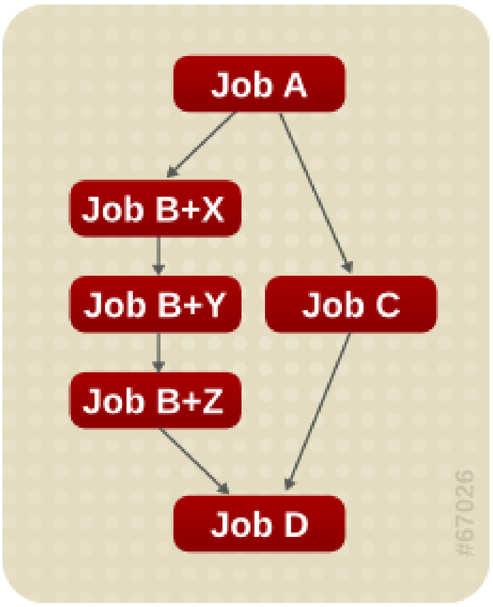
2. To nest the inner DAG inside an outer DAG, reference the inner DAG by using the **`SUBDAG EXTERNAL`** command:

```
# outer.dag
JOB   A   A_job.submit
SUBDAG   EXTERNAL B   inner.dag
JOB    C   C_job.submit
JOB    D   D_job.submit
PARENT   A   CHILD B,C
PARENT   B,C   CHILD D
```

3. If a job fails within a nested DAG, a rescue file will be generated for the inner DAG. The failure of the inner DAG results in the failure of the outer DAG, which will result in a rescue file also being created for the outer DAG. When the outer DAG file is resubmitted using **`condor_submit_dag`**, its rescue file will be run, which in turn will lead to the rescue file of the inner DAG also being run.

Splicing DAGs

Within nested DAGs, each individual DAG is managed by a dedicated **`condor_dagman`** process. This additional overhead can potentially put a strain on machine resources. An alternative is to use *splicing* instead of nesting. Splicing includes an external DAG definition inside another. The included nodes become part of a larger DAG that can all be managed by a single **`condor_dagman`** process. If one DAG fails, there will be a single rescue file that represents the state of all node jobs in the spliced DAG.



1. To create a spliced DAG, use the **`SPLICE`** in the following syntax:

```
SPLICE splice name DAG file name
```

2. A typical spliced submit description file would look something like this:

```
# big.dag
JOB   A   A_job.submit
SPLICE   B   inner.dag
JOB    C   C_job.submit
JOB    D   D_job.submit
PARENT   A   CHILD B,C
PARENT  B,C  CHILD D
```

DAGMan applies implicit scoping to the names of the spliced nodes inside the new DAG. It uses a + character between the splice name and the original DAG job name.

Job submission templates

Environment variables can be used to create job submission templates. Templates can be customized to provide details such as executable names and arguments.

1. Reference environment variables in a submission file using the **`$ENV`** syntax. If not done already, ensure that the variables to be used have been set in your Bash shell as follows:

```

$ export MYEXE=”/bin/sleep”

$ export MYARGS=”10”
```

2. Create the job submission file, referencing the environment variables:

```

# the dag job node file: dag_job.sub
executable      = $ENV(MYEXE)
arguments       = $ENV(MYARGS)
output          = dags/out/dag_job.out.$(cluster)
error           = dags/err/dag_job.err.$(cluster)
# log path can't use macro
log             = dags/log/diamond_dag.log
universe        = vanilla
notification    = NEVER
should_transfer_files = true
when_to_transfer_output = on_exit
queue
```

Application Program Interfaces (APIs)

The MRG Grid Web Service (WS) API is a tool for application developers to be able to interact with the system. The web interface allows jobs to be submitted and managed, and also offers a two-phase commit mechanism for reliability and fault-tolerance.

The MRG Grid daemons communicate using the SOAP XML protocol. An application using this protocol needs to contain code that can handle the communication. The XML Web services description language (WDSL) required by MRG Grid is included in the distribution, and can be found at **`$(RELEASE_DIR)/lib/webservice`**. The WSDL must be run through a toolkit to produce the language-specific routines required for communication.

19.1. Using the MRG Grid API

The application can be compiled as follows:

1. Condor must be configured to enable responses to SOAP calls. The WS interface listens on the **`condor_schedd`** daemon's command port. To obtain a list of all the the **`condor_schedd`** daemons in the pool that have a WS interface, use this command at the shell prompt:

   ```
   $ condor_status -schedd -constraint "HasSOAPInterface=?=TRUE"
   ```

2. To determine the port number to use:

   ```
   $ condor_status -schedd -constraint "HasSOAPInterface=?=TRUE" -l | grep MyAddress
   ```

3. To authorize access to the SOAP client, it is also important to set the **`ALLOW_SOAP`** and **`DENY_SOAP`** configuration variables.

Transactions

All applications that use the API to interact with the **`condor_schedd`** daemon use *transactions*. The lifetime of a transaction is limited by the API, and can be further limited by the client application or the **`condor_schedd`** daemon.

Transactions are controlled by methods. They are initiated with a **`beginTransaction()`** method and completed with either a **`commitTransaction()`** or an **`abortTransaction()`** method.

Some operations will have access to more information when they are performed within a transaction. As an example of this, a **`getJobAds()`** query would have access to information about pending jobs within the transaction. Because these jobs are not committed they would not be visible outside of the transaction. However, transactions are designed to be *ACID* (Atomic, Consistent, Isolated, and Durable). For this reason, information outside of a transaction should not be queried in order to make a decision within the transaction.

If required, the API can also accept null transactions. A null transaction can be created by inserting the programming language's equivalent of *`null`* in place of the transaction identifier. In a SOAP message, the following line achieves this:

```
<transaction xsi:type="ns1:Transaction" xsi:nil="true"/>
```

Submitting jobs

A job must be described with a ClassAd. The job ClassAd is then submitted to the **condor_schedd** within a transaction using the **submit()** method. To simplify the creation of a job ClassAd, the **createJobTemplate()** method can be called. This method returns a ClassAd structure that can then be modified to suit.

Important

For jobs that will be executed on Windows platforms, explicitly set the job ClassAd **NTDomain** attribute. The owner of the job will authenticate to this NT domain. This attribute is required but is not set by the **createJobTemplate()** function.

Necessary parts of the job ClassAd are the *ClusterId* and *ProcId* attributes, which uniquely identify the cluster and the job. When the **newCluster()** method is called, it is assigned a *ClusterId*. Every job submitted is then assigned a *ProcId*, starting at *0* and incrementing by one for every job. When **newCluster()** is called again, it is assigned the next *ClusterId* and the job numbering starts again at *0*.

This example demonstrates the *ClusterId* and *ProcId* attributes.

The following list contains an ordered set of method calls, showing the assigned *ClusterId* and *ProcId* values:

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*

Example 19.1. Demonstrating the *ClusterId* and *ProcId* attributes

There is always a chance that a call to **submit()** will fail. Mostly this occurs when the job is in the queue but something required by the job has not been sent and the job will not be able to be run succesfully. Sending the information required could potentially resolve this problem. To assist in determining what requirements a job has, the **`discoverJobRequirements()`** method can be called with a job ClassAd, and will return with a list of requirements for the job.

File transfer

Often, a job submission 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`** is running. The executable and input files must be sent directly to the **`condor_schedd`** daemon and placed in a spool location. This can be achieved with the **`declareFile()`** and **`sendFile()`** methods.

The **`declareFile()`** and **`sendFile()`** methods work together to transfer files to the **`condor_schedd`**. The **`declareFile()`** method causes **`condor_schedd`** to check if the file exists in the spool location. This prevents sending a file that already exists. The **`sendFile()`** method then sends the required file, or parts of a file, as base64 encoded data.

The **`declareFile()`** method requires the name of the file and its size in bytes. It also accepts optional information that relates to the hash (encryption) information for the file. When the hash type is specified as *`NOHASH`*, the **`condor_schedd`** daemon can not reliably determine if the file exists.

Retrieving files is most useful when a job is completed. When a job is completed and waiting to be removed, the **`listSpool()`** method provides a list of all the files for that job in the spool location. The **`getFile()`** method then retrieves a file.

Once the **`closeSpool()`** method has been called, the **`condor_schedd`** daemon removes the job from the queue and the spool files are no longer available. There is no requirement for the application to invoke the **`closeSpool()`** method, which results in jobs potentially remaining in the queue forever. The configuration variable **`SOAP_LEAVE_IN_QUEUE`** can help to mitigate this problem. It is a boolean value, and when it evaluates to *`False`*, the job will be removed from the queue, and its information moved into the history log.

This example demonstrates the use of the **`SOAP_LEAVE_IN_QUEUE`** configuration variable

The following line inserted in the configuration file will result in a job being removed from the queue once it has been completed for 24 hours:

```
SOAP_LEAVE_IN_QUEUE = ((JobStatus==4) && ((ServerTime - CompletionDate) < (60 * 60 * 24)))
```

Example 19.2. Use of the **`SOAP_LEAVE_IN_QUEUE`** configuration variable

19.2. Methods

Method	Description	Parameters	Return Value
`beginTransaction`	Begin a transaction.	*`duration`* - The expected duration of the transaction.	If the function succeeds, the return value is *`SUCCESS`* and contains the new transaction.
`commitTransaction`	Commits a transaction.	*`transaction`* - The transaction to be committed.	If the function succeeds, the return value is *`SUCCESS`*.

Method	Description	Parameters	Return Value
`abortTransaction`	Abort a transaction.	*`transaction`* - The transaction to be aborted.	If the function succeeds, the return value is *`SUCCESS`*.
`extendTransaction`	Request an extension in duration for a specific transaction.	*`transaction`* - The transaction to be extended and *`duration`* - The duration of the extension.	If the function succeeds, the return value is *`SUCCESS`* and contains the transaction with the extended duration.

Table 19.1. Methods for transaction management

`beginTransaction`

Begin a transaction. For example:

```
StatusAndTransaction beginTransaction(int duration);
```

`commitTransaction`

Commits a transaction. For example:

```
Status commitTransaction(Transaction transaction);
```

`abortTransaction`

Abort a transaction. For example:

```
Status abortTransaction(Transaction transaction);
```

`extendTransaction`

Request an extension in duration for a specific transaction. For example:

```
StatusAndTransaction extendTransaction( Transaction transaction, int duration);
```

Example 19.3. Examples of methods for transaction management

Method	Description	Parameters	Return Value
`submit`	Submit a job.	*`transaction`* - The transaction in which the submission takes place; *`clusterId`* - The cluster identifier; *`jobId`* - The job identifier; *`jobAd`* - The ClassAd describing the job.	If the function succeeds, the return value is *`SUCCESS`* and contains the transaction with the job requirements.

Method	Description	Parameters	Return Value
`createJobTemplate`	Request a job ClassAd, given some of the job requirements. This ClassAd will be suitable for use when submitting the job.	*`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; *`command`* - The command to execute once the job has started; *`arguments`* - The command-line arguments for *`command`*; *`requirements`* - The requirements expression for the job. *`type`* can be any one of the following: *`VANILLA = 5`*, *`SCHEDULER = 7`*, *`MPI = 8`*, *`GRID = 9`*, *`JAVA = 10`*, *`PARALLEL = 11`*, *`LOCALUNIVERSE = 12`* or *`VM = 13`*.	If the function succeeds, the return value is *`SUCCESS`*.
`discoverJobRequirements`	Discover the requirements of a job, given a ClassAd.	*`jobAd`* - The ClassAd of the job.	If the function succeeds, the return value is *`SUCCESS`* and contains the job requirements.

Table 19.2. Methods for job submission

`submit`

Submit a job. For example:

```
StatusAndRequirements submit(Transaction transaction, int clusterId, int jobId, ClassAd
 jobAd);
```

`createJobTemplate`

Request a job ClassAd, given some of the job requirements. This ClassAd will be suitable for use when submitting the job. For example:

```
StatusAndClassAd createJobTemplate(int clusterId, int jobId, String owner, UniverseType type,
 String command, String arguments, String requirements);
```

`discoverJobRequirements`

Discover the requirements of a job, given a ClassAd. For example:

```
StatusAndRequirements discoverJobRequirements( ClassAd jobAd);
```

Example 19.4. Examples of methods for job submission

Method	Description	Parameters	Return Value
`declareFile`	Declare a file to be used by a job.	*`transaction`* - The transaction in which the file is declared; *`clusterId`* - The cluster identifier; *`jobId`* - The 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; *`hash`* - An optionally zero-length string encoding of the file hash. *`hashType`* can be either *`NOHASH`* or *`MD5HASH`*	If the function succeeds, the return value is *`SUCCESS`*.
`sendFile`	Send a file that a job may use.	*`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; *`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.	If the function succeeds, the return value is *`SUCCESS`*.
`getFile`	Get a file from a job's spool.	*`transaction`* - An optionally nullable transaction, this call does not need to occur in a transaction; *`clusterId`* - The cluster in which to search; *`jobId`* - The job identifier the file is	If the function succeeds, the return value is *`SUCCESS`* and contains the file or a sub-section of the file as defined by offset and length.

Method	Description	Parameters	Return Value
		associated with; *`name`* - The name of the file to retrieve; *`offset`* - The starting offset within the file being retrieved; *`length`* - The length from the offset to retrieve.	
`closeSpool`	Close a job's spool. All the files in the job's spool can be deleted.	*`transaction`* - An optionally nullable transaction, 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.	If the function succeeds, the return value is *`SUCCESS`*.
`listSpool`	List the files in a job's spool.	*`transaction`* - An optionally nullable transaction, this call does not need to occur in a transaction; *`clusterId`* - The cluster in which to search; *`jobId`* - The job identifier to search for.	If the function succeeds, the return value is *`SUCCESS`* and contains a list of files and their respective sizes.

Table 19.3. Methods for file transfer

`declareFile`

Declare a file to be used by a job. For example:

```
Status declareFile(Transaction transaction, int clusterId, int jobId, String name, int size,
 HashType hashType, String hash);
```

`sendFile`

Send a file that a job can use. For example:

```
Status sendFile(Transaction transaction, int clusterId, int jobId, String name, int offset,
 Base64 data);
```

`getFile`

Get a file from a job's spool. For example:

```
StatusAndBase64 getFile(Transaction transaction, int clusterId, int jobId, String name, int
 offset, int length);
```

`closeSpool`

Close a job's spool. All the files in the job's spool can be deleted. For example:

```
Status closeSpool(Transaction transaction, int clusterId, int jobId);
```

`listSpool`

List the files in a job's spool. For example:

```
StatusAndFileInfoArray listSpool(Transaction transaction, int clusterId, int jobId);
```

Example 19.5. Examples of methods for file transfer

Method	Description	Parameters	Return Value
`newCluster`	Create a new job cluster.	*`transaction`* - The transaction in which this cluster is created.	If the function succeeds, the return value is *SUCCESS* and contains the cluster ID.
`removeCluster`	Remove a job cluster, and all the jobs within it.	*`transaction`* - An optionally nullable transaction, this call does not need to occur in a transaction; *`clusterId`* - The cluster to remove; *`reason`* - The reason for the removal.	If the function succeeds, the return value is *SUCCESS*.
`newJob`	Creates a new job within the most recently created job cluster.	*`transaction`* - The transaction in which this job is created; *`clusterId`* - The cluster identifier of the most recently created cluster.	If the function succeeds, the return value is *SUCCESS* and contains the job ID.
`removeJob`	Remove a job, regardless of the job's state.	*`transaction`* - An optionally nullable transaction, 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.	If the function succeeds, the return value is *SUCCESS*.

Method	Description	Parameters	Return Value
`holdJob`	Put a job into the Hold state, regardless of the job's current state.	*`transaction`* - An optionally nullable transaction, 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.	If the function succeeds, the return value is *SUCCESS*.
`releaseJob`	Release a job that has been in the Hold state.	*`transaction`* - An optionally nullable transaction, 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.	If the function succeeds, the return value is *SUCCESS*.
`getJobAds`	Find an array of job ClassAds.	*`transaction`* - An optionally nullable transaction, this call does not need to occur in a transaction; *`constraint`* - A string constraining the number of ClassAds to return.	If the function succeeds, the return value is *SUCCESS* and contains all job ClassAds matching the given constraint.
`getJobAd`	Finds a specific job ClassAd.	*`transaction`* - An optionally nullable transaction, this call	If the function succeeds, the return value is *SUCCESS* and

Method	Description	Parameters	Return Value
		does not need to occur in a transaction; *`clusterId`* - The cluster in which to search; *`jobId`* - The job identifier to search for.	contains the requested job ClassAd.
`requestReschedule`	Request a **`condor_reschedule`** from the **`condor_schedd`** daemon.		If the function succeeds, the return value is *`SUCCESS`*.

Table 19.4. Methods for job management

`newCluster`

Create a new job cluster. For example:

```
StatusAndInt newCluster(Transaction transaction);
```

`removeCluster`

Remove a job cluster, and all the jobs within it. For example:

```
Status removeCluster(Transaction transaction, int clusterId, String reason);
```

`newJob`

Creates a new job within the most recently created job cluster. For example:

```
StatusAndInt newJob(Transaction transaction, int clusterId);
```

`removeJob`

Remove a job, regardless of the job's state. For example:

```
Status removeJob(Transaction transaction, int clusterId, int jobId, String reason, boolean
 forceRemoval);
```

`holdJob`

Put a job into the Hold state, regardless of the job's current state. For example:

```
Status holdJob(Transaction transaction, int clusterId, int jobId, string reason, boolean
 emailUser, boolean emailAdmin, boolean systemHold);
```

`releaseJob`

Release a job that has been in the Hold state. For example:

```
Status releaseJob(Transaction transaction, int clusterId, int jobId, String reason, boolean
 emailUser, boolean emailAdmin);
```

getJobAds

Find an array of job ClassAds. For example:

```
StatusAndClassAdArray getJobAds(Transaction transaction, String constraint);
```

requestReschedule

Request a **condor_reschedule** from the **condor_schedd** daemon. For example:

```
Status requestReschedule();
```

Example 19.6. Examples of methods for job management

Method	Description	Parameters	Return Value
insertAd		*type* - The type of ClassAd to insert; *ad* - The ClassAd to insert. *type* can be any one of: *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* or *GENERIC-AD-TYPE*.	If the function succeeds, the return value is *SUCCESS*.
queryStartdAds	Search for **condor_startd** ClassAds.	*constraint* - A string constraining the number ClassAds to return.	A list of all the **condor_startd** ClassAds matching the given constraint.
queryScheddAds	Search for **condor_schedd** ClassAds.	*constraint* - A string constraining the number ClassAds to return.	A list of all the **condor_schedd** ClassAds matching the given constraint.
queryMasterAds	Search for **condor_master** ClassAds.	*constraint* - A string constraining the number ClassAds to return.	A list of all the **condor_master** ClassAds matching the given constraint.
querySubmittorAds	Search for submitter ClassAds.	*constraint* - A string constraining the number ClassAds to return.	A list of all the submitter ClassAds matching the given constraint.
queryLicenseAds	Search for license ClassAds.	*constraint* - A string constraining the	A list of all the license ClassAds matching the given constraint.

Method	Description	Parameters	Return Value
		number ClassAds to return.	
`queryStorageAds`	Search for storage ClassAds.	*`constraint`* - A string constraining the number ClassAds to return.	A list of all the storage ClassAds matching the given constraint.
`queryAnyAds`	Search for any ClassAds.	*`constraint`* - A string constraining the number ClassAds to return.	A list of all the ClassAds matching the given constraint.

Table 19.5. Methods for ClassAd management

`insertAd`

For example:

```
Status insertAd(ClassAdType type, ClassAdStruct ad);
```

`queryStartdAds`

Search for **`condor_startd`** ClassAds. For example:

```
ClassAdArray queryStartdAds(String constraint);
```

`queryScheddAds`

Search for **`condor_schedd`** ClassAds. For example:

```
ClassAdArray queryScheddAds(String constraint);
```

`queryMasterAds`

Search for **`condor_master`** ClassAds. For example:

```
ClassAdArray queryMasterAds(String constraint);
```

`querySubmittorAds`

Search for submitter ClassAds. For example:

```
ClassAdArray querySubmittorAds(String constraint);
```

`queryLicenseAds`

Search for license ClassAds. For example:

```
ClassAdArray queryLicenseAds(String constraint);
```

`queryStorageAds`

Search for storage ClassAds. For example:

```
ClassAdArray queryLicenseAds(String constraint);
```

`queryAnyAds`

Search for any ClassAds. For example:

```
ClassAdArray queryAnyAds(String constraint);
```

Example 19.7. Examples of methods for ClassAd management

Method	Description	Return Value
`getVersionString`	Determine the Condor version.	Returns the Condor version as a string.
`getPlatformString`	Determine the platform information.	Returns the platform information as string.

Table 19.6. Methods for version information

`getVersionString`

Determine the Condor version. For example:

```
StatusAndString getVersionString();
```

`getPlatformString`

Determine the platform information. For example:

```
StatusAndString getPlatformString();
```

Example 19.8. Examples of methods for version information

Many methods return a status, *Table 19.7, “StatusCode return values”* lists the possible return values:

Value	Identifier	Definition
0	*SUCCESS*	No errors returned.
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, or can not be found.
5	*UNKNOWNFILE*	The specified file does not exist, or can not be found.
6	*INCOMPLETE*	The request is incomplete.
7	*INVALIDOFFSET*	The specified offset is invalid.
8	*ALREADYEXISTS*	For this job, the specified file already exists

Table 19.7. StatusCode return values

Frequently Asked Questions

Q: How do I download MRG Grid?

A: MRG Grid is available through the Red Hat Network. For full instructions on downloading and installing MRG Grid, read the *MRG Grid Installation Guide* available from the *Red Hat Enterprise MRG documentation page*[1].

Q: What platforms are supported?

A: MRG Grid is supported under most recent versions of Red Hat Enterprise Linux. Full information is available from the *Red Hat Enterprise MRG hardware page*[2].

Q: Can I access the source code?

A: Yes! The source code is made available in the source RPM distributed by Red Hat. MRG Grid source code is distributed under the *Apache ASL 2.0 license*[3].

Q: MRG Grid sends me too much email. What should I do with it all?

A: You should not ignore all the mail sent to you, but you can dramatically reduce the amount you get. When jobs are submitted, ensure they contain the following line:

```
Notification = Error
```

This will make sure that you only receive an email if an error has occurred. Note that this means you will not receive emails when a job completes successfully.

Q: My job starts but exits right away with *`signal 9`*. What's wrong?

A: This error occurs most often when a shared library is missing. If you know which file is missing, you can re-install it on all machines that might execute the job. Alternatively, re-link your program so that it contains all the information it requires.

Q: None or only some of my jobs are running, even though there's resources available in the pool. How can I fix this?

A: Firstly, you will need to discover where the problem lies. Try these steps to work out what is wrong:

1. Run **`condor_q -analyze`** and **`condor_q -better`** to check the output they give you
2. Look at the User Log file. This is the file that you specified as **`log =`** ***`path/to/filename.log`*** in the submit file. From this file you should be able to tell if the jobs are starting to run, or if they are exiting before they begin.
3. Look at the SchedLog on the submit machine after it has performed the negotiation for the user. If a user doesn't have a high enough enough priority to access more resources, then this log will contain a message that says *`Lost priority, no more jobs`*.

4. Check the ShadowLog on the submit machine for warnings or errors. If jobs are successfully being matched with machines, they still might be failing when they try to execute. This can be caused by file permission problems or similar errors.

5. Look at the NegotiatorLog during the negotiation for the user. Look for messages about priority or errors such as *`No more machines`*.

Another common problem that will stop jobs running is if the submit machine does not have adequate swap space. This will produce an error in the **`SCHEDD_LOG`** file:

```
[date] [time] Swap space estimate reached! No more jobs can be run!
[date] [time] Solution: get more swap space, or set RESERVED_SWAP = 0
[date] [time] 0 jobs matched, 1 jobs idle
```

The amount of swap space on the submit machine is calculated by the system. Serious errors can occur in a situation where a machine has a lot of physical memory and little or no swap space. Because physical memory is not considered, Condor might calculate that it has little or no swap space, and so it will not run the submitted jobs.

You can check how much swap space has been calculated as being available, by running the following command from the shell prompt:

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

If the value in the output is *`0`*, then you will need to tell the system that it has some swap space. This can be done in two ways:

1. Configure the machine with some more actual swap space; or

2. Disable the check. Define the amount of reserved swap space for the submit machine as *`0`*, and change the **`RESERVED_SWAP`** configuration variable to *`0`*. You will need to perform **`condor_restart`** on the submit machine to pick up the changes.

Q: I submitted a job, but now my requirements expression has extra things in it that I didn't put there. How did they get there and why do I need them?

A: This occurs automatically, and are extensions that are required by Condor. This is a list of the things that are automatically added:

- If *`arch`* and *`opsys`* are not specified in the submit description file, they will be added. It will insert the same platform details as the machine from which the job was submitted.

- The expression **`Memory * 1024 > ImageSize`** is automatically added. This makes sure that the job runs on a machine with at least as much physical memory as the memory footprint of the job.

- If the **`Disk >= DiskUsage`** is not specified, it will be added. This makes sure that the job will only run on a machine with enough disk space for the job's local input and output.

- A pool administrator can request that certain expressions are added to submit files. This is done using the following configuration variables:

- **APPEND_REQUIREMENTS**
- **APPEND_REQ_VANILLA**
- **APPEND_REQ_STANDARD**

Q: What signals get sent to my jobs when they are pre-empted or killed, or when I remove them from the queue? Can I tell Condor which signals to send?

A: The signal jobs are sent can be set in the submit description file, by adding either of the following lines:

```
remove_kill_sig = SIGWHATEVER
```

```
kill_sig = SIGWHATEVER
```

If no signal is specified, the **SIGTERM** signal will be used. In the case of a hard kill, the **SIGKILL** signal is sent instead.

Q: Why does the time output from **condor_status** appear as *[?????]*?

A: Collecting time data from an entire pool of machines can cause errant timing calculations if the system clocks of those machines differ. If a time is calculated as negative, it will be displayed as *[?????]*. This can be fixed by synchronizing the time on all machines in the pool, using a tool such as NTP (Network Time Protocol).

Q: Condor commands are running very slowly. What is going on?

A: Some Condor commands will react slowly if they expect to find a **condor_collector** daemon, but can not find one. If you are not running a **condor_collector** daemon, change the **COLLECTOR_HOST** configuration variable to nothing:

```
COLLECTOR_HOST=
```

Q: If I submit jobs under NFS, they fail a lot. What's going on?

A: If the directory you are using when you run **condor_submit** is automounted under NFS (Network File System), Condor might try to unmount the volume before the job has completed.

To fix the problem, use the **initialdir** command in your submit description file with a reference to the stable access point. For example, if the NFS automounter is configured to mount a volume at **/a/myserver.company.com/vol1/user** whenever the directory **/home/user** is accessed, add this line to the submit description file:

```
initialdir = /home/user
```

Q: Why is my Java job completing so quickly?

A: The java universe executes the Java program's **`main()`** method and waits for it to return. When it returns, Condor considers your job to have been completed. This can happen inadvertantly if the **`main()`** method is starting threads for processing. To avoid this, ensure you **`join()`** all threads spawned in the **`main()`** method.

Q: Are there any special configuration macros I can use?

A: Yes. Use this command at the shell prompt to find out what they are:

```
$ env CONDOR_CONFIG=ONLY_ENV condor_config_val -dump
```

Q: I can submit a job through the web service interface of condor using the SOAP API, and then remove the job from the pool using **`condor_rm`**. But when I check that the job has been removed, **`condor_q`** reports the status as *X*. How do I remove the job completely?

A: Jobs are marked as completed using the **`closeSpool`** method. If the **`closeSpool`** is not invoked, jobs can remain in the queue forever. Use the **`SOAP_LEAVE_IN_QUEUE`** configuration variable to fix this problem. A good option is to set the **`SOAP_LEAVE_IN_QUEUE`** variable to invoke the **`closeSpool`** method once the job has been completed for 24 hours, like this:

```
SOAP_LEAVE_IN_QUEUE = ((JobStatus==4) && ((ServerTime - CompletionDate) < (60 * 60 *
 24)))
```

Q: I tried to submit a Linux job to a Linux central manager using a node running Windows. Why didn't it work?

A: Submitting Linux jobs from a node running Windows to a Linux central manager is currently not supported. Submit the job from a machine running Linux instead.

Q: My log files contain errors saying *`PERMISSION DENIED`*. What does that mean?

A: This can happen if the configuration variables **`ALLOW_*`** and **`DENY_*`** are not configured correctly. Check these parameters and set **`ALLOW_*`** and **`DENY_*`** as appropriate.

Q: What happens if the central manager crashes?

A: If the central manager crashes, jobs that are already running will continue as normal. Queued jobs will remain in the queue but will not begin running until the central manager is restarted and begins matchmaking again.

Q: The condor daemons are running, but I get no output when I run **`condor_status`**. What is wrong?

A: Check the collector log. You should see a message similar to this:

```
DaemonCore: PERMISSION DENIED to host 128.105.101.15:9618 for command 0
 (UPDATE_STARTD_AD)
```

This type of error is caused when permissions are configured correctly. Try the following:

- Ensure that DNS inverse lookup works on your machines (when you type in an IP address, you machine can find the domain name). If it is not working, either fix the DNS problem itself, or set the **DEFAULT_DOMAIN_NAME** setting in the configuration file

- Use numeric IP addresses instead of domain names when setting the **ALLOW_WRITE** and **DENY_WRITE** configuration macros

- If the problem is caused by being too restrictive, try using wildcards when defining the address. For example, instead of using:

  ```
  ALLOW_WRITE = condor.your.domain.com
  ```

  try using:

  ```
  ALLOW_WRITE = *.your.domain.com
  ```

Q: How do I stop my job moving to different CPUs?

A: You will need to define which slot you want the job to run on. You can do this using either **numactl** or **taskset**. If you are running jobs from within your own program, use **sched_setaffinity** and **pthred_{,attr_}setaffinity** to achieve the same result.

Q: I have a High Availability setup, but sometimes the **schedd** keeps on trying to start but exits with a *status 0*. Why is this happening?

A: In an High-Available Scheduler setup with 2 nodes (Node A and Node B), Condor will start on Node A and brings up the **schedd**, before it starts on Node B. On node B, the **schedd** continually attempts to start and exits with *status 0*.

This can be caused by the two nodes using different HA **schedd** names. In this case, the **schedd** on Node B will continually try to start, but will not be able to because of lock conflicts.

This problem can be solved by using the same name for the **schedd** on both nodes. This will make the **schedd** on Node B realize that one is already running, and it doesn't need to start. Change the **SCHEDD_NAME** configuration entry on both nodes so that the name is identical.

Note that this configuration will allow other schedulers to run on other nodes besides the HA **SCHEDD_NAME**. So you can have HA (on two nodes) and other **schedd**s elsewhere.

Q: When I use a custom kill signal, the **condor_startd** crashes. Why does this happen?

A: When you try to kill a job with a custom signal, it can sometimes cause a race condition to occur between the starter and the startd. This happens when the startd communicates with the starter using **procd**. The startd will always wait the value specified in the *killing_timeout* parameter before hard-killing the starter. However, by default the starter will wait for the value specified in the *killing_timeout-1* configuration variable before attempting to hard-kill the job. This means that it is sometimes possible for the startd to be attempting to hard-kill the starter, while the starter is cleaning up and exiting. It causes the starter to stop communicating with the **procd**, which makes the startd suffer a communication failure, and then crash.

This can be handled in two ways:

1. Set **`STARTD.USE_PROCD = FALSE`** and **`STARTER.USE_PROCD = FALSE`** in the configuration settings. This is the most reliable way to handle the situation.

2. All jobs that use a custom kill signal should have **`kill_sig_timeout`** set to a reasonable time in the submit description file. This will require adjustment, as the timing can be dependent on the jobs running, and the load on the startd. Also, **`kill_sig_timeout`** cannot be a larger value than *`killing_timeout-1`*.

More Information

Reporting Bugs

Follow these instructions to enter a bug report:

1. You will need a *Bugzilla*[1] account. You can create one at *Create Bugzilla Account*[2].
2. Once you have a Bugzilla account, log in and click on *Enter A New Bug Report*[3].
3. You will need to identify the product (Red Hat Enterprise MRG), the version (1.3), and whether the bug occurs in the software (component=grid) or in the documentation (component=Grid_Installation_Guide).

Further Reading

Red Hat Enterprise MRG and MRG Grid Product Information
: *http://www.redhat.com/mrg*

MRG Grid User Guide and other Red Hat Enterprise MRG manuals
: *http://docs.redhat.com/docs/en-US/index.html*

Condor Manual
: *http://www.cs.wisc.edu/condor/manual/*

Red Hat Knowledgebase
: *https://access.redhat.com/knowledge/search*

Appendix A. Configuration options

This section describes individual variables used to configure all parts of the MRG Grid system. General information about the configuration files and their syntax can be found in *Chapter 2, Configuration*

A.1. Pre-defined configuration macros

MRG Grid provides pre-defined configuration macros to help simplify configuration. These settings are determined automatically and cannot be overwritten.

`FULL_HOSTNAME`
: The fully qualified hostname of the local machine (domain name and hostname)

`HOSTNAME`
: The hostname of the local machine

`IP_ADDRESS`
: The local machine's IP address as an ASCII string

`TILDE`
: The full path to the home directory of the user running MRG Grid, if the user exists on the local machine. By default, this will be the *`condor`* user.

Subsystems

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 MRG Grid system. Some possible subsystem names are:

- **`STARTD`**
- **`SCHEDD`**
- **`MASTER`**
- **`COLLECTOR`**
- **`NEGOTIATOR`**
- **`KBDD`**
- **`SHADOW`**
- **`STARTER`**
- **`GRIDMANAGER`**
- **`HAD`**
- **`REPLICATION`**
- **`JOB_ROUTER`**

A.2. Static pre-defined configuration macros

These settings are determined automatically and cannot be overwritten.

`ARCH`
: Defines the string used to identify the architecture of the local machine to MRG Grid. This allows jobs to be submitted for a given platform and MRG Grid will force them to run on the correct machines

`OPSYS`
: Defines the string used to identify the operating system of the local machine to MRG Grid. If it is not defined in the configuration file, MRG Grid will automatically insert the operating system of the current machine as determined by the **`uname`** command

`UNAME_ARCH`
: The architecture as reported by the **`uname`** command's *`machine`* field

`UNAME_OPSYS`
: The operating system as reported by the **`uname`** command's *`sysname`* field

`PID`
: The process ID of the daemon or tool

`PPID`
: The process ID of the daemon or tool's parent process

`USERNAME`
: The name of the user running the daemon or tool. For daemons started as the root user, but running under another user, that username will be used instead of root

A.3. System Wide Configuration File Variables

These settings affect all parts of the MRG Grid system.

`FILESYSTEM_DOMAIN`
: Defaults to the fully qualified hostname of the current machine.

`UID_DOMAIN`
: Defaults to the fully qualified hostname of the current machine it is evaluated on.

`COLLECTOR_HOST`
: The host name of the machine where the **`condor_collector`** is running for the pool. **`COLLECTOR_HOST`** must be defined for the pool to work properly.

  This setting can also be used to specify the network port of the **`condor_collector`**. The port is separated from the host name by a colon. To set the network port to 1234, use the following syntax:

```
COLLECTOR_HOST = $(CONDOR_HOST):1234
```

  If no port is specified, the default port of 9618 is used.

CONDOR_VIEW_HOST
: The host name of the machine where the **CondorView** server is running. This service is optional, and requires additional configuration to enable it. If **CONDOR_VIEW_HOST** is not defined, no **CondorView** server is used.

RELEASE_DIR
: The full path to the MRG Grid release directory, which holds the **bin**, **etc**, **lib** and **sbin** directories. There is no default value for **RELEASE_DIR**.

BIN
: The directory where user-level programs are installed.

LIB
: The directory where libraries used to link jobs for MRG Grid's standard universe are stored.

LIBEXEC
: The directory where support commands are placed. Do not add this directory to a user or system-wide path.

INCLUDE
: The directory where header files are placed.

SBIN
: The directory where system binaries and administrative tools are installed. The directory defined at **SBIN** should also be in the path of users acting as administrators.

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)**, in this format:

```
LOCAL_DIR = $(TILDE)
```

On machines with a shared file system, where the directory is shared among all machines in your pool, use the **$(HOSTNAME)** macro and have a directory with many sub-directories, one for each machine in your pool. For example:

```
LOCAL_DIR = $(tilde)/hosts/$(hostname)
```

or:

```
LOCAL_DIR = $(release_dir)/hosts/$(hostname)
```

LOG
: The directory where each daemon writes its log files. The names of the log files themselves are defined with other macros, which require the **$(LOG)** macro.

SPOOL
: The directory where files used by **condor_schedd** are stored, including the job queue file and the initial executables of any jobs that have been submitted. If a given machine executes jobs but does not submit them, it does not require a **SPOOL** directory.

EXECUTE

The scratch directory for the local machine. The scratch directory is used as the destination for 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 is specified. If a given machine submits jobs but does not execute them, it does not require an **EXECUTE** directory. To customize the execute directory independently for each batch slot, use **SLOTx_EXECUTE**.

REQUIRE_LOCAL_CONFIG_FILE

A boolean value that defaults to true. This will cause MRG Grid to exit with an error if any file listed in **LOCAL_CONFIG_FILE** cannot be located. If the value is set to false, MRG Grid will ignore any local configuration files that cannot be located and continue. If **LOCAL_CONFIG_FILE** is not defined, and **REQUIRE_LOCAL_CONFIG_FILE** has not been explicitly set to false, an error will be caused.

CONDOR_IDS

The User ID (UID) and Group ID (GID) for Condor daemons to use when run by the root user. This value can also be set using the **CONDOR_IDS** environment variable. The syntax is:

```
CONDOR_IDS = UID.GID
```

To set a UID of 1234 and a GID of 5678, use the following setting:

```
CONDOR_IDS = 1234.5678
```

If **CONDOR_IDS** is not set and the daemons are run by the root user, MRG Grid will search for a condor user on the system, and use that UID and GID.

CONDOR_ADMIN

An email address for MRG Grid to send messages about any errors that occur in the pool, such as a daemon failing.

CONDOR_SUPPORT_EMAIL

The email address to be included in the footer of all email sent out by MRG Grid. The footer reads:

```
Email address of the local MRG Grid administrator: admin@example.com
```

If this setting is not defined, MRG Grid will use the address specified in **CONDOR_ADMIN**.

MAIL

The full path to a text based email client, such as **/bin/mail**. The email client must be able to accept mail messages and headers as standard input (**STDIN**) and use the **-s** command to specify a subject for the message. On all platforms, the default shipped with MRG Grid should work. This setting will only need to be changed if the installation is in a non-standard location. The **condor_schedd** will not function unless **MAIL** is defined.

RESERVED_SWAP

The amount (in megabytes) of memory swap space reserved for use by the machine. MRG Grid will stop initializing processes if the amount of available swap space falls below this level. The default value is 5MB.

`RESERVED_DISK`
: The amount (in megabytes) of disk space reserved for use by the machine. When reporting, MRG Grid will subtract this amount from the total amount of available disk space. The default value is 0MB (zero megabytes).

`LOCK`
: MRG Grid creates lock files in order to synchronize access to various log files. If the local Condor directory is not on a local partition, be sure to set the **`LOCK`** entry to avoid problems with file locking.

  The user and group that MRG Grid runs as need to have write access to the directory that contains the lock files. If no value for **`LOCK`** is provided, the value of **`LOG`** is used.

`HISTORY`
: The location of the history file, which stores information about all jobs that have completed on a given machine. This setting is used by **`condor_schedd`** to append information, and **`condor_history`** the user-level program used to view the file. The default value is **`$(SPOOL)/history`**. If not defined, no history file will be kept.

`ENABLE_HISTORY_ROTATION`
: A boolean value that defaults to true. When false, the history file will not be rotated, and the history will continue to grow in size until it reaches the limits defined by the operating system. The rotated files are stored in the same directory as the history file. Use **`MAX_HISTORY_LOG`** to define the size of the file and **`MAX_HISTORY_ROTATIONS`** to define the number of files to use when rotation is enabled.

`MAX_HISTORY_LOG`
: Defines the maximum size (in bytes) for the history file, before it is rotated. Default value is 20,971,520 bytes (20MB). This parameter is only used if history file rotation is enabled.

`MAX_HISTORY_ROTATIONS`
: Defines how many files to use for rotation. Defaults to 2. In this case, there may be up to three history files at any one time - two backups and the history file that is currently being written. The oldest file will removed first on rotation.

`MAX_JOB_QUEUE_LOG_ROTATIONS`
: The job queue database file is periodically rotated in order to save disk space. This option controls how many rotated files are saved. Defaults to 1. In this case, there may be up to two history files at any one time - the backup which has been rotated out of use, and the history file that is currently being written. The oldest file will be removed first on rotation.

`NO_DNS`
: A boolean value that defaults to false. When true, MRG Grid constructs hostnames automatically using the machine's IP address and **`DEFAULT_DOMAIN_NAME`**.

`DEFAULT_DOMAIN_NAME`
: The domain name for the machine. This value is appended to the hostname in order to create a fully qualified hostname. This value should be set in the global configuration file, as MRG Grid can depend on knowing this value in order to locate the local configuration files. The default value is an example, and must be changed to a valid domain name. This variable only operates when **`NO_DNS`** is set to true.

EMAIL_DOMAIN
: Defines the domain to use for email. If a job is submitted and the user has not specified *notify_user* in the submit description file, MRG Grid will send any email about that job to *username*@**UID_DOMAIN**. If all the machines share a common UID domain, but email to this address will not work, you will need to define the correct domain to use. In many cases, you can set **EMAIL_DOMAIN** to **FULL_HOSTNAME**.

CREATE_CORE_FILES
: A boolean value that is undefined by default, in order to allow the default operating system value to take precedence. If set to true, the Condor daemons will create core files in the **LOG** directory in the case of a segmentation fault (segfault). When set to false no core files will be created. When left undefined, it will retain the setting that was in effect when the Condor daemons were started. Core files are used primarily for debugging purposes.

ABORT_ON_EXCEPTION
: A boolean value that defaults to false. When set to true MRG Grid will abort on a fatal internal exception. If **CREATE_CORE_FILES** is also true, MRG Grid will create a core file when an exception occurs.

Q_QUERY_TIMEOUT
: The amount of time (in seconds) that **condor_q** will wait when trying to connect to **condor_schedd**, before causing a timeout error. Defaults to 20 seconds.

DEAD_COLLECTOR_MAX_AVOIDANCE_TIME
: For pools where High Availability is in use. Defines the maximum time (in seconds) to wait in between checks for a failed primary **condor_collector** daemon. If connections to the dead daemon take very little time to fail, new query attempts become more frequent. Defaults to 3600 (1 hour).

NETWORK_MAX_PENDING_CONNECTS
: The maximum number of simultaneous network connection attempts. **condor_schedd** can 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. Defaults to 80% of the process file descriptor limit, except on Windows operating systems, where the default is 1600.

WANT_UDP_COMMAND_SOCKET
: A boolean value that defaults to true. When true, Condor daemons will create a UDP command socket in addition to the required TCP command socket. When false, only the TCP command socket will be created. If you modify this setting, you will need to restart all Condor daemons.

MASTER_INSTANCE_LOCK
: The name of the lock file to prevent multiple **condor_master** daemons from starting. This is useful when using shared file systems like NFS, where the lock files exist on a local disk. Defaults to **$(LOCK)/InstanceLock**. The **$(LOCK)** macro can be used to specify the location of all lock files, not just the **condor_master** instance lock. If **$(LOCK)** is undefined, the master log itself will be locked.

SHADOW_LOCK
: The lock file to be used for access to the **ShadowLog** file. It must be a separate file from the **ShadowLog**, since the ShadowLog might be rotated and access will need to be synchronized across rotations. This macro is defined relative to the **$(LOCK)** macro.

`LOCAL_QUEUE_BACKUP_DIR`
: The directory to use to back up the local queue. This directory must be located on a non-network filesystem.

`LOCAL_XACT_BACKUP_FILTER`
: Defines whether or not to back up transactions based on whether or not the commit was successful. When set to *`ALL`* local transaction backups will always be kept. When set to *`NONE`* local transaction backups will never be kept. When set to *`FAILED`* local transaction backups will be kept for transactions that have failed to commit.

  To retain backups, **`LOCAL_QUEUE_BACKUP_DIR`** must be set to a valid directory and **`LOCAL_XACT_BACKUP_FILTER`** must be set to something other than *`NONE`*.

`X_CONSOLE_DISPLAY`
: The name of the display that the condor kbdd daemon should monitor. Defaults to *`:0.0`*.

A.4. Logging configuration variables

These variables control logging. Many of these variables apply to each of the possible subsystems. In each case, replace the word *`SUBSYSTEM`* with the name of the appropriate subsystem.

***`SUBSYSTEM`*`_LOG`**
: The name of the log file for a given subsystem. For example, **`STARTD_LOG`** gives the location of the log file for the **`condor_startd`** daemon.

`MAX_`*`SUBSYSTEM`*`_LOG`
: The maximum size a log file is allowed to grow to, in bytes. Each 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, thus the maximum space devoted to logging for any one program will be twice the maximum length of its log file. A value of *`0`* specifies that the file may grow without bounds. Deafults to 1MB.

`TRUNC_`*`SUBSYSTEM`*`_LOG_ON_OPEN`
: When *`TRUE`*, the log will be restarted with an empty file every time the program is run. When *`FALSE`* new entries will be appended. Defaults to *`FALSE`*.

***`SUBSYSTEM`*`_LOCK`**
: Specifies the lock file used to synchronize additions to the log file. It must be a separate file from the ***`SUBSYSTEM`*`_LOG`** file, since that file can be rotated and synchronization should occur 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 setting is for Windows platforms only. When *`TRUE`* logs are able to be locked using a mutex instead of by file locking. This can correct problems on Windows platforms where processes starve waiting for a lock on a log file. Defaults to *`TRUE`* on Windows platforms. Always set to *`FALSE`* on Unix platforms.

`ENABLE_USERLOG_LOCKING`
: When *`TRUE`* the job log specified in the submit description file is locked before being written to. Defaults to *`TRUE`*.

`TOUCH_LOG_INTERVAL`

The time interval between daemons creating (using the **`touch`** command) log files, in seconds. 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`

Formatting of the current time at the start of each line in the log files. When *TRUE*, Unix Epoch Time is used. When *FALSE*, the time is printed in the local timezone using the syntax:

```
[Month]/[Day]/[Year] [Hour]:[Minute]:[Second]
```

Defaults to *FALSE*.

***`SUBSYSTEM`*`_DEBUG`**

The Condor daemons are all capable of producing different levels of output. All daemons default to **`D_ALWAYS`**. This logs all messages. Settings are a comma or space-separated list of these values:

- **`D_ALL`**

  The most verbose logging option. This setting generates an extremely large amount of output.

- **`D_FULLDEBUG`**

  Verbose output. Only very frequent log messages for very specific debugging purposes are excluded.

- **`D_DAEMONCORE`**

  Logs messages that specific to DaemonCore, such as timers the daemons have set and the commands that are registered.

- **`D_PRIV`**

  Logs messages about privilege state switching.

- **`D_COMMAND`**

  With this flag set, any daemon that uses DaemonCore will print out a log message whenever a command is received. 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.

- **`D_LOAD`**

  The **`condor_startd`** records the load average on the machine where it is running. Both the general system load average, and the load average being generated by MRG Grid activity 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`** subsystem.

- **`D_KEYBOARD`**

  Logs messages related to the values for remote and local keyboard idle times. This flag only affects the **`condor_startd`** subsystem.

- **`D_JOB`**

  Logs the contents of any job ClassAd that the **`condor_schedd`** sends to claim the **`condor_startd`**. This flag only affects the **`condor_startd`** subsystem.

- **`D_MACHINE`**

  Logs the contents of any machine ClassAd that the **`condor_schedd`** sends to claim the **`condor_startd`**. This flag only affects the **`condor_startd`** subsystem.

- **`D_SYSCALLS`**

  Logs remote syscall requests and return values.

- **`D_MATCH`**

  Logs messages for every match performed by the **`condor_negotiator`**.

- **`D_NETWORK`**

  All daemons will log a message on every TCP accept, connect, and close, and on every UDP send and receive.

- **`D_HOSTNAME`**

  Logs verbose messages explaining how host names, domain names and IP addresses have been resolved.

- **`D_SECURITY`**

  Logs messages regarding secure network communications. Includes messages about negotiation of a socket authentication mechanism, management of a session key cache, and messages about the authentication process.

- **`D_PROCFAMILY`**

  Logs messages regarding management of families of processes. A process family is defined as a process and all descendents of that process.

- **`D_ACCOUNTANT`**

  Logs messages regarding the computation of user priorities.

- **`D_PROTOCOL`**

  Log messages regarding the protocol for the matchmaking and resource claiming framework.

- **`D_PID`**

  This flag is used to change the formatting of all log messages that are printed. If **`D_PID`** is set, the process identifier (PID) of the process writing each line to the log file will be recorded.

- **`D_FDS`**

  This flag is used to change the formatting of all log messages that are printed. If **`D_FDS`** is set, the file descriptor that the log file was allocated will be recorded.

`ALL_DEBUG`
: Used to make all subsystems share a debug flag. For example, to turn on all debugging in all subsystems, set **`ALL_DEBUG = D_ALL`**.

`TOOL_DEBUG`
: Uses the same values (debugging levels) as ***`SUBSYSTEM`*`_DEBUG`** to describe the amount of debugging information sent to *`STDERR`* for Condor tools.

`SUBMIT_DEBUG`
: Uses the same values (debugging levels) as ***`SUBSYSTEM`*`_DEBUG`** to describe the amount of debugging information sent to *`STDERR`* for **`condor_submit`**.

***`SUBSYSTEM_[LEVEL]`*`_LOG`**
: This is the name of a log file for messages at a specific debug level for a specific subsystem. If the debug level is included in ***`SUBSYSTEM`*`_DEBUG`**, then all messages of this debug level will be written both to the ***`SUBSYSTEM`*`_LOG`** file and the ***`SUBSYSTEM_[LEVEL]`*`_LOG`** file.

`MAX_`*`SUBSYSTEM_[LEVEL]`*`_LOG`
: Similar to **`MAX_`*`SUBSYSTEM`*`_LOG`**.

`TRUNC_`*`SUBSYSTEM_[LEVEL]`*`_LOG_ON_OPEN`
: Similar to **`TRUNC_`*`SUBSYSTEM`*`_LOG_ON_OPEN`**.

`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 it is not defined.

`MAX_EVENT_LOG`
: 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`* allows the file to grow continuously. Defaults to 1MB.

`EVENT_LOG_USE_XML`
: When *`TRUE`*, events are logged in XML format. Defaults to *`FALSE`*.

`EVENT_LOG_JOB_AD_INFORMATION_ATTRS`
: A comma-separated list of job ClassAd attributes. When evaluated, these values form a new event of **`JobAdInformationEvent`**. This new event is placed in the event log in addition to each logged event.

A.5. DaemonCore Configuration Variables

`ALLOW...`
: All macros that begin with either **`ALLOW`** or **`DENY`** are settings for host-based security.

`ENABLE_RUNTIME_CONFIG`
: The **`condor_config_val`** tool has an option **`-rset`** for dynamically setting run time configuration values (which only effect 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, 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 (described below). This setting 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 configuration files. Because of the potential security implications of this feature, by default, Condor daemons will not honor these requests. To use this functionality, 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 (described below). This setting 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 MRG Grid 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 ***`SUBSYSTEM`*`_SETTABLE_ATTRS`** are settings used to restrict the configuration values that can be changed using the **`condor_config_val`** command.

`SHUTDOWN_GRACEFUL_TIMEOUT`
Determines how long to allow daemons to try to gracefully shut down before they do a hard shutdown. It is defined in terms of seconds. The default is 1800 (30 minutes).

***`SUBSYSTEM`*`_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 *`SUBSYSTEM`* with the appropriate subsystem string.

***SUBSYSTEM*_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**).

***SUBSYSTEM*_ATTRS**

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 ***SUBSYSTEM*** with the appropriate subsystem string.

NOTE: The **condor_kbdd** does not send ClassAds, 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.

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

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. If they are not defined, they default to *FALSE*.

NOTE: This functionality does not work in conjunction with the high availability feature. 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.

`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 running. If the parent does not receive this message after a proscribed period, it assumes that the child process is hung. It then kills and restarts the child process. 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 messages at an interval of one third of this value.

***`SUBSYSTEM`*`_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 a message from the **`condor_schedd`** before killing it.

`NOT_RESPONDING_WANT_CORE`

A boolean parameter with a default value of false. This parameter is for debugging purposes on UNIX systems, and controls the behavior of the parent process when it 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 **`CREATE_CORE_FILES`** enabled, the child process will then generate a core dump.

`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.

A.6. Network-Related Configuration File Entries

`BIND_ALL_INTERFACES`

For systems with multiple network interfaces, if this configuration setting is *`False`*, network sockets will only bind to the IP address specified with **`NETWORK_INTERFACE`** (described below). If set to *`True`*, the default value, MRG Grid will listen on all interfaces. However, currently MRG Grid 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.

`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.

SUBSYSTEM**_MAX_FILE_DESCRIPTORS**

This setting is identical to **MAX_FILE_DESCRIPTORS**, but it only applies to a specific subsystem. If the subsystem-specific setting is unspecified, **MAX_FILE_DESCRIPTORS** is used.

MAX_FILE_DESCRIPTORS

This specifies the maximum number of file descriptors the Condor daemons are allowed 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.

After adjusting this configuration variable, restart MRG Grid to pick up the changes. Note that if MRG Grid is running as root, the limit can only be increased above the hard limit (on maximum open files) on inherited files.

NETWORK_INTERFACE

For systems with multiple network interfaces, if this configuration setting is not defined, all network sockets will bind to the first interface found. To bind to a specific network interface other than the first one, this **NETWORK_INTERFACE** should be set to the IP address to use. When **BIND_ALL_INTERFACES** is set to *True* (the default), this setting simply controls what IP address a given host will advertise.

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 the *PrivateIpAddr* attribute from the classified ad. 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 *PrivateIpAddr* 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.

PRIVATE_NETWORK_INTERFACE

In 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.

The **PRIVATE_NETWORK_INTERFACE** defines what IP address a given multi-homed machine should use for the private network. 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.

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 for this setting.

HIGHPORT

Specifies an upper limit of given port numbers to use, to create a range of available port numbers. If this macro is not explicitly specified, then the port numbers available will not be restricted, and system-assigned port numbers will be used. For this macro to work, both *HIGHPORT* and *LOWPORT* (given below) must be defined.

LOWPORT

Specifies a lower limit of given port numbers, to create a range of available port numbers. If this macro is not explicitly specified, then the port numbers available will not be restricted, and system-assigned port numbers will be used. 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 use on incoming connections (listening ports), to create a range of available 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 use on incoming connections (listening ports), to create a range of available 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 use on outgoing connections, to create a range of available 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 a upper limit of given port numbers for use on outgoing connections, to create a range of available 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

This setting defaults to **False**. If your site needs to use TCP connections to send ClassAd updates to your collector set to this to *True*. At this time, this setting only affects the main **condor_collector** for the site. If enabled, also define **COLLECTOR_SOCKET_CACHE_SIZE** at the central manager, so that the collector will accept TCP connections for updates, and will keep them open for reuse. 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**).

`TCP_UPDATE_COLLECTORS`
The list of collectors which will be updated with TCP instead of UDP. If not defined, no collectors use TCP instead of UDP.

`SUBSYSTEM`**`_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.

`NET_REMAP_ENABLE`
A boolean variable, that when defined to *`True`*, enables a network remapping service. 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`
This boolean variable determines the behavior of the **`condor_master`** with GCB enabled. It defaults to *`True`*.

When **`MASTER_WAITS_FOR_GCB_BROKER`** is *`True`*; if there is no GCB broker working when the **`condor_master`** starts, or if communications with a GCB broker fail, the **`condor_master`** waits while attempting to find a working GCB broker.

When **`MASTER_WAITS_FOR_GCB_BROKER`** is *`False`*; if no GCB broker is working when the **`condor_master`** starts the **`condor_master`** fails and exits without restarting. If the **`condor_master`** has successfully communicated with a GCB broker at start-up but the communication fails, the **`condor_master`** kills all its children, exits, and restarts.

A.7. Shared File System Configuration File Macros

`UID_DOMAIN`

A further check attempts to assure that the submitting machine can not lie about its **`UID_DOMAIN`**. The submit machine's claimed value for **`UID_DOMAIN`** is compared 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, the job will run as user *`nobody`*. For example; a job submission to the pool from *`flippy.example.com`*, claiming a **`UID_DOMAIN`** of *`flipper.example.com`*, will run the job as the user *`nobody`*. Because of this verification, **`UID_DOMAIN`** must be a real domain name.

Also see **`SOFT_UID_DOMAIN`** below for information about a further check that are performed before running a job as a specific user.

Note: An administrator could set **`UID_DOMAIN`** to `*`. This will match all domains, but it produces a serious security risk. It is not recommended.

An administrator can also leave **`UID_DOMAIN`** undefined. This will force jobs to always run as user *`nobody`*. 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 e-mail is sent about a job, it uses the address *`user@$(UID_DOMAIN)`*. If **`UID_DOMAIN`** is undefined, the e-mail is sent to *`user@submitmachinename`*.

`TRUST_UID_DOMAIN`

When a job is about to launch, MRG Gridit ensures that the **`UID_DOMAIN`** of a given submit machine is a substring of that machine's fully-qualified host name. The default setting of **`TRUST_UID_DOMAIN`** is *`False`* as this test is a security precaution. At some sites, however, there may be multiple UID spaces that do not clearly correspond to Internet domain names and in these cases administrators may wish to use names which are not substrings of the host names to describe the UID domains.

In order for this measure to work, the **`UID_DOMAIN`** check must not occur. If the **`TRUST_UID_DOMAIN`** setting is *`True`*, this test will not occur, and whatever **`UID_DOMAIN`** is presented by the submit machine will be trusted.

`SOFT_UID_DOMAIN`

A boolean variable that defaults to *`False`* when not defined. When a job is run as a particular user (instead of as user *`nobody`*), it verifies that the UID given for the user is in the password file and 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. For this check not to occur, set this configuration variable to *`True`*. The job will then be run under the user's UID.

`SLOTx_USER`

The name of a user to be used 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`*. *`x`* is an integer associated with slots.

`STARTER_ALLOW_RUNAS_OWNER`

This is 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 **`SLOTx_USER`** or *`nobody`* (*`False`*). In Unix, this defaults to *`True`*. In Windows, it defaults to *`False`*. The job ClassAd may also contain an attribute **`RunAsOwner`** which is logically ANDed with the starter's boolean value. Under Unix, if the job does not specify it, this attribute defaults to *`True`*. Under

Windows, it defaults to *`False`*. In Unix, if the **`UID_DOMAIN`** of the machine and job do not match, there is no possibility to run the job as the owner so this setting has no effect.

`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 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 **`SLOTx_USER`** to be a different account for each slot, and no non-condor processes will ever be run by these accounts, then this pattern should match the names of all **`SLOTx_USER`** accounts.

Jobs run under a dedicated execute account are reliably tracked, whereas other jobs might spawn processes that are not detected. 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, all processes owned by the dedicated execution account will attempt to be killed.

For 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"
```

`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 (however, it often is).

This implementation is not ideal; machines may share some file systems but not others. There is currently 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.

Note that if you do not set **`FILESYSTEM_DOMAIN`**, it 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.

`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.

A.8. `condor_master` Configuration File Macros

`DAEMON_LIST`

This macro determines what daemons the **`condor_master`** will start and monitor. The list is a comma or space separated list of subsystem names. For example:

```
DAEMON_LIST = MASTER, STARTD, SCHEDD
```

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.

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
```

SUBSYSTEM

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 *SUBSYSTEM* with the appropriate subsystem string as defined in previous sections.

***DAEMONNAME*_ENVIRONMENT**

For each subsystem defined in **DAEMON_LIST**, you may specify changes to the environment that daemon is started with by setting ***DAEMONNAME*_ENVIRONMENT**, where *DAEMONNAME* is the name of a daemon listed in **DAEMON_LIST**. It should use the same syntax for specifying the environment as the environment specification in a **condor_submit** file. For example, if you wish to redefine the **TMP** and **CONDOR_CONFIG** environment variables seen by the **condor_schedd**, you could place the following in the config file:

```
SCHEDD_ENVIRONMENT = "TMP=/new/value CONDOR_CONFIG=/special/config"
```

When the **condor_schedd** was started by the **condor_master**, it would see the specified values of **TMP** and **CONDOR_CONFIG**.

SUBSYSTEM_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, 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 *SUBSYSTEM* with the appropriate subsystem string.

PREEN

In addition to the daemons defined in **DAEMON_LIST**, the **condor_master** also starts up a special process called **condor_preen** to clean out junk files that have been left behind. This macro determines where the **condor_master** finds the **condor_preen** binary. This macro can be commented out to prevent **condor_preen** from running.

`PREEN_ARGS`

This macro controls how **`condor_preen`** behaves by allowing the specification of command-line arguments. This macro works as ***`SUBSYSTEM`*****`_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. The **`-m`** switch will instruct MRG Grid to send e-mail about files that should be removed. **`-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`* the **`condor_master`** will exit as soon as it starts. This setting is useful if the boot scripts for your entire pool are centralized but you do not want MRG Grid to run on certain machines. This entry is most effectively used in a file in the local configuration directory, not a global configuration file.

`START_DAEMONS`

This macro is similar to the **`START_MASTER`** macro described above. This macro, however, does not force the **`condor_master`** to exit; instead preventing it from starting any of the daemons listed in the **`DAEMON_LIST`**. The daemons may be started later 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 it. 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_BACKOFF_CONSTANTand **MASTER_*name*_BACKOFF_CONSTANT**

When a daemon crashes, **condor_master** uses an exponential back off delay before restarting it (see the "Backoff Delays" section below for details 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 "Backoff Delays" section below for details 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 "Backoff Delays" section below for details 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.

***SUBSYSTEM*_USERID**

Specifies the userid under which a subsystem should run.

Backoff Delays

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 crash. 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 $t = 9 + 2^{0}$, 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*) to the power the number of times the daemon has restarted, and add **`MASTER_BACKOFF_CONSTANT`** (which defaults to *9*). Thus:

- 1st crash: n = 0, so: $t = 9 + 2^{0} = 9 + 1$ = 10 seconds
- 2nd crash: n = 1, so: $t = 9 + 2^{1} = 9 + 2$ = 11 seconds
- 3rd crash: n = 2, so: $t = 9 + 2^{2} = 9 + 4$ = 13 seconds

And so on...

- 9th crash: n = 8, so: $t = 9 + 2^{8} = 9 + 256$ = 265 seconds

until, after 13 crashes, it would be:

- 13th crash: n = 12, so: $t = 9 + 2^{12} = 9 + 4096$ = 4105 seconds

This last result is higher than the **`MASTER_BACKOFF_CEILING`**, which defaults to *`3600`*, so the daemon would 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). Should the daemon stay alive for the time set in **`MASTER_RECOVER_FACTOR`** (defaults to 5 minutes), the count of how many restarts this daemon has performed is reset to *`0`*.

Important

The default settings work quite well, you will probably not need to change them.

`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, one will be appended, 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. The string will not be modified 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. Since the assumption is 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, the **STARTD_NAME** should be set to uniquely identify the **condor_startd** daemons.

If a Condor daemon **master, schedd or startd**) has been given a unique name, all 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 A.5, "DaemonCore Configuration Variables"* under ***SUBSYSTEM*_ATTRS**.

MASTER_DEBUG
This macro is described in *Section A.4, "Logging configuration variables"* as ***SUBSYSTEM*_DEBUG**.

MASTER_ADDRESS_FILE
This macro is described in *Section A.5, "DaemonCore Configuration Variables"* as ***SUBSYSTEM*_ADDRESS_FILE**.

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_masters** 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 MRG Grid 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, MRG Grid 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 it 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. Process groups help to track the descendants of processes that have been created. This can cause problems when MRG Grid is run under another job execution system.

A.9. `condor_startd` Configuration File Macros

`START`

A boolean expression that, when *`True`*, indicates that the machine is willing to start running a 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 a running job to be suspended. The machine may still be claimed, but the job makes no further progress, and no load is generated on the machine.

`PREEMPT`

A boolean expression that, when *`True`*, causes a currently running job to be stopped.

`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_SUB_CODE`

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 a previously suspended job to continue executed.

`KILL`

A boolean expression that, when *`True`*, causes the execution of a currently running job to stop without delay.

RANK
A floating point value that is used to compare potential jobs. A larger value for a specific job ranks that job above others with lower values for **RANK**.

WANT_SUSPEND
A boolean expression that, when *True*, will evaluate the **SUSPEND** expression.

WANT_VACATE
A boolean expression that, when *True*, defines that a preempted 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*.

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)**.

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. This macro is defined in terms of seconds and defaults to *300* (5 minutes).

UPDATE_OFFSET
An integer value representing the number of seconds 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 *0*.

The example configuration:

```
startd.UPDATE_INTERVAL = 300
startd.UPDATE_OFFSET   = $RANDOM_INTEGER(0,300)
```

would cause 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) implements the expected policy that there is no retirement time.

CLAIM_WORKLIFE

If provided, this expression specifies the number of seconds after which a claim will stop accepting additional jobs.

Once the negotiator gives a **schedd** a claim to a slot the **schedd** will, by default, keep running jobs on that slot (as long as it has jobs with matching requirements) without returning the slot to the unclaimed state and renegotiating for machines. The solution is to use **CLAIM_WORKLIFE** to force the claim to stop running additional jobs after a certain amount of time. Once **CLAIM_WORKLIFE** expires, any existing job may continue to run as usual, but once it finishes or is preempted, the claim is closed.

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). 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.

This macro may be useful if you want to force periodic renegotiation of resources without preemption having to occur.

MAX_CLAIM_ALIVES_MISSED

This setting controls how many keep alive messages can be missed by the **condor_startd** before it considers a resource claim by a **condor_schedd** no longer valid. The default is *6*.

The **condor_schedd** sends periodic keep alive updates to each **condor_startd**. 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. As keep alive messages are sent via UDP and are sometimes dropped by the network, the **condor_startd** has some tolerance for missed keep alive messages. If a few keep alive messages are not recieved, the **condor_startd** will not immediately release the claim. This macro sets the number of missed messages that will be tolerated.

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 systems where "mouse" is used 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 need to either add it or change this macro to point to the right device.

`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_SENDS_ALIVES`

A boolean value that defaults to *`False`*, such that the **`condor_schedd`** daemon sends keep alive signals to the **`condor_startd`** daemon. When *`True`*, the **`condor_startd`** daemon sends keep alive signals to the **`condor_schedd`** daemon, reversing the direction. This may be useful if the **`condor_startd`** daemon is on a private network or behind a firewall.

`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, the ClaimId will be written into a file in the **`LOG`** directory called **`.startd_claim_id.slot`*`X`***, 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 the **`.slotX`** portion of the file name will be automatically appended.

`NUM_CPUS`

Important

This option is intended for advanced users and is disabled by default.

This macro is 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 automatic detection of CPUs.

Enabling this can allow multiple jobs to run on a single-CPU machine by having that machine treated like an SMP machine with multiple CPUs, each running different jobs. Alternatively, an SMP machine may advertise more slots than it has CPUs. However, lying in this manner will affect the performance of the jobs, since now multiple jobs will will compete with each other on the same CPU.

If lying about the CPUs in a given machine, you should use the **`STARTD_ATTRS`** setting to advertise the fact in the machine's ClassAd. This will allow jobs submitted in the pool to specify if they do not want to be matched with machines that are only offering these fractional CPUs.

Note: This setting cannot be changed with a simple reconfigure, either by sending a **SIGHUP** or by using the **condor_reconfig** command. To change this macro you must restart the **condor_startd** daemon. The command is:

```
condor_restart -subsystem startd
```

MAX_NUM_CPUS
An integer value used as a ceiling for the number of CPUs detected 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 hyper-threaded processors are treated. 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, the amount of physical memory available on your machine will be automatically detected. Define **MEMORY** to state how much physical memory (in MB) your machine has, overriding the automatic value.

RESERVED_MEMORY
By default, all the physical memory of the machine as considered available to be used by jobs. If **RESERVED_MEMORY** is defined, this value is subtracted from the amount of memory advertised 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 *Section A.8, “**condor_master** Configuration File Macros ”* for defaults and composition of valid Condor daemon names.

RUNBENCHMARKS
Specifies when to run benchmarks. Benchmarks will be run when the machine is in the Unclaimed state and this expression evaluates to *True*. 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.

STARTD_RESOURCE_PREFIX

A string which specifies what prefix to give the unique resources that are advertised on SMP machines. The default value of this prefix is *slot*. This setting enables sites to define what string the **condor_startd** will use to name the individual resources on an SMP machine if they prefer to use something other than *slot*.

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 (that is slots that notice when there is 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 total 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).

This ensures the slot is available to jobs as soon as the **condor_startd** starts up (if the slot is configured to ignore keyboard activity), instead of having to wait for 15 minutes (which is the default time a machine must be idle before a job will start) or more.

If you do not want this boost, set the value to *0*. Increase this macro's value 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.

STARTD_SLOT_ATTRS

The list of ClassAd attribute names that should be shared across all slots on the same machine. 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
```

MAX_SLOT_TYPES

The maximum number of different slot types.This macro defaults to *10* (you should only need to change this setting if you define more than 10 separate slot types).

Note: this is the maximum number of different slot *types*, not of actual slots.

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**.

SLOT_TYPE_*N*_PARTITIONABLE

A boolean variable that defaults to *False*. When set to *True*, this slot permits dynamic slots.

NUM_SLOTS_TYPE_*N*

This macro controls how many of a given slot type are actually reported. There is no default.

NUM_SLOTS

This macro controls how many slots will be reported if your 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 pool. You cannot use this parameter to advertise more slots than there are CPUs on the machine. To do that, use **NUM_CPUS**.

ALLOW_VM_CRUFT

A boolean value that is set and used internally, currently defaulting to *True*. When *True*, MRG Grid 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*, it does not look for variables named with the previously used string VM after searching unsuccessfully for the string **SLOT**.

STARTD_CRON_NAME

Defines a logical name to be used in the formation of related configuration macro names. While not required, this macro makes other macros more readable and maintainable. A common example is:

```
STARTD_CRON_NAME = HAWKEYE
```

This example allows the naming of other related macros to contain the string "*HAWKEYE*" in their name.

STARTD_CRON_CONFIG_VAL

This configuration variable can be used to specify the **condor_config_val** program which the modules (jobs) should use to get configuration information from the daemon. If this is provided, a environment variable by the same name with the same value will be passed to all modules.

If **STARTD_CRON_NAME** is defined, then this configuration macro name is changed from **STARTD_CRON_CONFIG_VAL** to **$(STARTD_CRON_NAME)_CONFIG_VAL**. Example:

```
HAWKEYE_CONFIG_VAL = /usr/local/condor/bin/condor_config_val
```

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 cron modules produce output.

Important

Enabling this setting can greatly increase the network traffic in a pool, especially when many modules are executed or if they are run in short intervals.

There are three possible values for this setting:

`never`
: This default value causes the **`condor_startd`** to not automatically publish updates based on any cron modules. Instead, updates rely on the usual behavior for sending updates, which is periodic, based on the **`UPDATE_INTERVAL`** configuration setting, or whenever a given slot changes state.

`always`
: Causes the **`condor_startd`** to always send a new update to the **`condor_collector`** whenever any module exits.

`if_changed`
: Causes the **`condor_startd`** to only send a new update to the **`condor_collector`** if the output produced by a given module is different than the previous output of the same module. The only exception is the *`LastUpdate`* attribute (automatically set for all cron modules to be the timestamp when the module last ran), which is ignored when **`STARTD_CRON_AUTOPUBLISH`** is set to **`if_changed`**.

Be aware that **`STARTD_CRON_AUTOPUBLISH`** does not honor the **`STARTD_CRON_NAME`** setting described above. Even if **`STARTD_CRON_NAME`** is defined, **`STARTD_CRON_AUTOPUBLISH`** will have the same name.

`STARTD_CRON_JOBLIST`
: This configuration variable is defined by a white space separated list of job names (called modules) to run. Each of these is the logical name of the module. This name must be unique (no two modules may have the same name).

If **`STARTD_CRON_NAME`** is defined, then this configuration macro name is changed from **`STARTD_CRON_JOBLIST`** to **`$(STARTD_CRON_NAME)_JOBLIST`**.

`STARTD_CRON_`*`ModuleName`*`_PREFIX`
: Specifies a string which is prepended to all attribute names that the specified module generates. For example, if a prefix is set as "*`xyz_`*", and an individual attribute is named *`abc`*", the resulting attribute would be *`xyz_abc`*. Although it can be quoted the prefix can contain only alpha-numeric characters.

If **`STARTD_CRON_NAME`** is defined, then this configuration macro name is changed from **`STARTD_CRON_`*`ModuleName`*`_PREFIX`** to **`$(STARTD_CRON_NAME)_`*`ModuleName`*`_PREFIX`**.

`STARTD_CRON_`*`ModuleName`*`_EXECUTABLE`
: Used to specify the full path to the executable to run for this module. Note that multiple modules may specify the same executable (although they need to have different names).

If **`STARTD_CRON_NAME`** is defined, then this configuration macro name is changed from **`STARTD_CRON_`*`ModuleName`*`_EXECUTABLE`** to **`$(STARTD_CRON_NAME)_`*`ModuleName`*`_EXECUTABLE`**.

STARTD_CRON_*ModuleName*_PERIOD

The period specifies time intervals at which the module should be run. For periodic modules, this is the time interval that passes between starting the execution of the module. The value may be specified in seconds (append value with the character '*s*'), in minutes (append value with the character '*m*'), or in hours (append value with the character '*h*'). For example, *5m* starts the execution of the module every five minutes. If no character is appended to the value, seconds are used as a default. The minimum valid value of the period is 1 second.

For "Wait For Exit" mode, the value has a different meaning; in this case the period specifies the length of time after the module ceases execution before it is restarted.

If **STARTD_CRON_NAME** is defined, this configuration macro name is changed from **STARTD_CRON_*ModuleName*_PERIOD** to **$(STARTD_CRON_NAME)_*ModuleName*_PERIOD**.

STARTD_CRON_*ModuleName*_MODE

Used to specify the "Mode" in which the module operates. Legal values are "WaitForExit" and "Periodic" (the default).

The default "Periodic" mode is used for most modules. In this mode, the module is expected to be started by the **condor_startd** daemon, gather and publish its data, and then exit.

The "WaitForExit' mode is used to specify a module which runs in the "Wait For Exit" mode. In this mode, the **condor_startd** daemon interprets the "period" differently. In this case, it refers to the amount of time to wait after the module exits before restarting it. With a value of *1*, the module is kept running nearly continuously.

In general, "Wait For Exit" mode is for modules that produce a periodic stream of updated data, but it can be used for other purposes as well.

STARTD_CRON_*ModuleName*_RECONFIG

The "ReConfig" macro is used to specify whether a module can handle HUP signals, and should be sent a HUP signal, when the **condor_startd** daemon is reconfigured. The module is expected to reread its configuration at that time. A value of "True" enables this setting, and "False" disables it.

If **STARTD_CRON_NAME** is defined, then this configuration macro name is changed from **STARTD_CRON_*ModuleName*_RECONFIG** to:

```
$(STARTD_CRON_NAME)_ModuleName_RECONFIG.
```

STARTD_CRON_*ModuleName*_KILL

The "Kill" macro is applicable on for modules running in the "Periodic" mode. Possible values are "True" and "False" (the default).

If **STARTD_CRON_NAME** is defined, then this configuration macro name is changed from **STARTD_CRON_*ModuleName*_KILL** to **$(STARTD_CRON_NAME)_*ModuleName*_KILL**.

This macro controls the behavior of the **condor_startd** when it detects that the module's executable is still running when it is time to start the module for a run. If enabled, the **condor_startd** will kill and restart the process in this condition. If not enabled, the existing process is allowed to continue running.

STARTD_CRON_*ModuleName*_ARGS

The command line arguments to pass to the module to be executed.

If **STARTD_CRON_NAME** is defined, then this configuration macro name is changed from **STARTD_CRON_*ModuleName*_ARGS** to **$(STARTD_CRON_NAME)_*ModuleName*_ARGS**.

STARTD_CRON_*ModuleName_ENV*

The environment string to pass to the module. The syntax is the same as that of **DAEMONNAME_ENVIRONMENT**.

If **STARTD_CRON_NAME** is defined, then this configuration macro name is changed from **STARTD_CRON_*ModuleName*_ENV** to **$(STARTD_CRON_NAME)_*ModuleName*_ENV**.

STARTD_CRON_*ModuleName*_CWD

The working directory in which to start the module.

If **STARTD_CRON_NAME** is defined, then this configuration macro name is changed from **STARTD_CRON_*ModuleName*_CWD** to **$(STARTD_CRON_NAME)_*ModuleName*_CWD**.

STARTD_CRON_*ModuleName*_OPTIONS

A colon separated list of options. Not all combinations of options make sense; when a nonsense combination is listed, the last one in the list is followed.

If **STARTD_CRON_NAME** is defined, then this configuration macro name is changed from **STARTD_CRON_*ModuleName*_OPTIONS** to **$(STARTD_CRON_NAME)_*ModuleName*_OPTIONS**.

- The "WaitForExit" option enables the "Wait For Exit" mode (see above).
- The "ReConfig" option enables the "Reconfig" setting (see above).
- The "NoReConfig" option disables the "Reconfig" setting (see above).
- The "Kill" option enables the "Kill" setting (see above).
- The "NoKill" option disables the "Kill" setting (see above).

STARTD_CRON_JOBS

The list of the modules to execute. In *Hawkeye*, this is usually named **HAWKEYE_JOBS**. This configuration variable is defined by a white space or newline separated list of jobs (called modules) to run, where each module is specified using the format:

```
modulename:prefix:executable:period[:options]
```

Each of these fields can be surrounded by matching quote characters (single quote or double quote, but they must match). This allows colon and white space characters to be specified. For example, the following specifies an executable name with a colon and a space in it:

```
foo:foo_:"c:/some dir/foo.exe":10m
```

These individual fields are described below:

- **modulename**: The logical name of the module. This must be unique (no two modules may have the same name). See **STARTD_CRON_JOBLIST**.
- **prefix**: See **STARTD_CRON_*ModuleName*_PREFIX**.
- **executable**: See **STARTD_CRON_*ModuleName*_EXECUTABLE**.

- **`period`**: See **`STARTD_CRON_`*`ModuleName`*`_PERIOD`**.
- Several options are available. Using more than one of these options for one module does not make sense. If this happens, the last one in the list is followed. See **`STARTD_CRON_`*`ModuleName`*`_OPTIONS`**.
  - The "Continuous" option is used to specify a module which runs in continuous mode (as described above). See the "WaitForExit" and "ReConfig" options which replace "Continuous".

    This option is now deprecated, and its functionality has been replaced by the new "WaitForExit" and "ReConfig" options, which together implement the capabilities of "Continuous".
  - The "WaitForExit" option

    See the discussion of "WaitForExit" in **`STARTD_CRON_`*`ModuleName`*`_OPTIONS`** above.
  - The "ReConfig" option

    See the discussion of "ReConfig" in **`STARTD_CRON_`*`ModuleName`*`_OPTIONS`** above.
  - The "NoReConfig" option

    See the discussion of "NoReConfig" in **`STARTD_CRON_`*`ModuleName`*`_OPTIONS`** above.
  - The "Kill" option

    See the discussion of "Kill" in **`STARTD_CRON_`*`ModuleName`*`_OPTIONS`** above.
  - The "NoKill" option

    See the discussion of "NoKill" in **`STARTD_CRON_`*`ModuleName`*`_OPTIONS`** above.

  NOTE: The configuration file parsing logic will strip white space from the beginning and end of continuation lines. Thus, a job list like below will be misinterpreted and will not work as expected:

```
# Hawkeye Job Definitions
HAWKEYE_JOBS   =\
    JOB1:prefix_:$(MODULES)/job1:5m:nokill\
    JOB2:prefix_:$(MODULES)/job1_co:1h
HAWKEYE_JOB1_ARGS =-foo -bar
HAWKEYE_JOB1_ENV = xyzzy=somevalue
HAWKEYE_JOB2_ENV = lwpi=somevalue
```

  Instead, write this as below:

```
# Hawkeye Job Definitions
HAWKEYE_JOBS   =

# Job 1
HAWKEYE_JOBS = $(HAWKEYE_JOBS) JOB1:prefix_:$(MODULES)/job1:5m:nokill
HAWKEYE_JOB1_ARGS =-foo -bar
HAWKEYE_JOB1_ENV = xyzzy=somevalue

# Job 2
HAWKEYE_JOBS = $(HAWKEYE_JOBS) JOB2:prefix_:$(MODULES)/job2:1h
```

```
HAWKEYE_JOB2_ENV = lwpi=somevalue
```

STARTD_COMPUTE_AVAIL_STATS

A boolean that determines if the **condor_startd** computes resource availability statistics. The default is *False*.

If **STARTD_COMPUTE_AVAIL_STATS** = *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 (that is, 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.

JAVA

The full path to the Java interpreter (the Java Virtual Machine).

JAVA_MAXHEAP_ARGUMENT

An incomplete command line argument to the Java interpreter (the Java Virtual Machine) to specify the switch name for the Maxheap Argument. This is used to construct the maximum heap size for the Java Virtual Machine. For example, the value for the Sun JVM is *-Xmx*.

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.

`SLOTN_JOB_HOOK_`*`KEYWORD`*

The keyword used to define which set of hooks a particular compute slot should invoke. Note that the "*`N`*" in "**`SLOTN`**" should be replaced with the slot identification number, for example, on *`slot1`*, this setting would 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`*

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 these job hooks must explicitly define the keyword (and the corresponding hook paths).

`HOOK_FETCH_WORK`

The full path to the program to invoke whenever the **`condor_startd`** wants to fetch work. The actual configuration setting must be prefixed with a hook keyword. There is no default.

`HOOK_REPLY_CLAIM`

The full path to the program to invoke whenever the **`condor_startd`** finishes fetching a job and decides what to do with it. The actual configuration setting must be prefixed with a hook keyword. There is no default.

`HOOK_EVICT_CLAIM`

The full path to the program to invoke whenever the **`condor_startd`** needs to evict a fetched claim. The actual configuration setting must be prefixed with a hook keyword. There is no default.

`FetchWorkDelay`

An expression that defines the number of seconds that the **`condor_startd`** should wait after an invocation of **`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.

`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 the machine to be put into a low power state given by the evaluation of the expression. 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, Sleep (standby)
- "S2", "2": On Windows, Sleep (standby)
- "S3", "3", "RAM", "MEM": Sleep (standby)
- "S4", "4", "DISK", "HIBERNATE": Hibernate
- "S5", "5", "SHUTDOWN": 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 S*n*, 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 S*n* 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".

LINUX_HIBERNATION_METHOD

A string that can be used to override the default search used on Linux platforms to detect the hibernation method to use. 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.

A.10. condor_schedd Configuration File Entries

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 *True*. 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 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 MRG Grid that existed before the local universe to one that includes the local universe, but without using the newer configuration files.

START_SCHEDULER_UNIVERSE

A boolean value that defaults to *True*. 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 the **condor_schedd** ClassAd:

```
START_SCHEDULER_UNIVERSE = TotalSchedulerJobsRunning < 10
```

A.11. condor_starter Configuration File Entries

These settings affect the **condor_starter**.

JOB_RENICE_INCREMENT

When the **condor_starter** spawns a job, it can set a nice level. This is a mechanism that allows users to assign processes a lower priority. This can mean that those processes do not interfere with interactive use of the machine.

The integer value of the nice level is set by the **condor_starter** daemon just before each job runs. The range of allowable values are integers in the range of 0 to 19, with 0 being the highest priority and 19 the lowest. If the integer value is outside this range, then a value greater than 19 is auto-decreased to 19 and a value less than 0 is treated as 0. The default value is 10.

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 ShadowLog. It defaults to True.

STARTER_DEBUG

This setting refers to the level of information sent to the log. See *Section A.4, "Logging configuration variables"* for more information.

STARTER_UPDATE_INTERVAL

The number of seconds to wait between ClassAd updates. This value is sent to the **condor_startd** and **condor_shadow** daemons. Defaults to 300 seconds (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. If monitoring takes a long time, the **condor_starter** will monitor less frequently than specified. The default value is 0.1.

USER_JOB_WRAPPER

The full path to an executable or script. This macro is used to specify a wrapper script to handle the execution of all user jobs. If specified, the job will never be run directly. The program specified will be invoked instead. The command-line arguments passed to this program are the full path to the actual job to be executed, and all the command line parameters to pass to the job. This wrapper program will ultimately replace its image with the user job. This means that it must execute the user job, instead of forking it.

STARTER_JOB_ENVIRONMENT

Used to set the default environment that is inherited by jobs. It uses the same syntax as the environment settings in the job submit file. If the same environment variable is assigned by this macro and by the user in the submit file, the user settings takes precedence.

JOB_INHERITS_STARTER_ENVIRONMENT

A boolean value. When *TRUE*, jobs will inherit all environment variables from the **condor_starter**. When both the user job and **STARTER_JOB_ENVIRONMENT** define an environment variable, the user's job definition takes precedence. This variable does not apply to standard universe jobs. Defaults to *FALSE*

`STARTER_UPLOAD_TIMEOUT`

An integer value that specifies the number of seconds to wait when transferring files back to the submit machine, before declaring a network timeout. 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. The default value is 300 seconds (5 minutes).

`ENFORCE_CPU_AFFINITY`

A boolean value. When *FALSE*, the CPU affinity of jobs and their descendents is not enforced. When *TRUE*, CPU affinity will be maintained, and finely tuned affinities can be specified using **`SLOTX_CPU_AFFINITY`**. Defaults to *FALSE*

`SLOTX_CPU_AFFINITY`

A comma separated list of cores. The specified cores are those to which a job running on **`SLOTX`** will show affinity. This setting will work only if **`ENFORCE_CPU_AFFINITY`** is set to *TRUE*.

`SCHEDD_CLUSTER_INITIAL_VALUE`

Specifies the first cluster ID number to be assigned. Defaults to *1*. If the job cluster ID reaches the value set by **`SCHEDD_CLUSTER_MAXIMUM_VALUE`** and wraps around, the job cluster ID will be reset to the value of **`SCHEDD_CLUSTER_INITIAL_VALUE`**.

If the **`job_queue.log`** file is removed, cluster IDs will be assigned starting from **`SCHEDD_CLUSTER_INITIAL_VALUE`** after system restart.

`SCHEDD_CLUSTER_MAXIMUM_VALUE`

An upper bound on the job cluster ID. If this parameter is set to a value (*M*), the maximum job cluster ID assigned to any job will be (*M-1*). When the maximum ID is reached, job IDs will wrap around back to **`SCHEDD_CLUSTER_INITIAL_VALUE`**. The default value is *0*, which will not set a maximum cluster ID.

Warning

It is important to ensure that the number of jobs in the queue at any one time is less than the value of this parameter. If too many jobs are queued at once, duplicate cluster IDs could be assigned. Additionally, it is important that a job is never submitted with a cluster ID the same as an already running job. Duplicate cluster IDs will result in a corrupted job queue.

`SCHEDD_CLUSTER_INCREMENT_VALUE`

Specifies the increment to use when assigning new cluster ID numbers. Defaults to *1*.

For example, if **`SCHEDD_CLUSTER_INITIAL_VALUE`** is set to *2*, and **`SCHEDD_CLUSTER_INCREMENT_VALUE`** is set to *2*, the cluster ID numbers will be *{2, 4, 6, ...}*.

A.12. Example configuration files

This section contains complete default configuration files.

This is the default global configuration file. It is located at **`/etc/condor/condor_config`** and is usually the same for all installations. Do not change this file. To customize the configuration, edit files in the local configuration directory instead.

```
########################################################################
########################################################################
###                                                                  ###
###  N O T I C E:    D O   N O T   E D I T   T H I S    F I L E     ###
###                                                                  ###
###      Customization should be done via the LOCAL_CONFIG_DIR.     ###
###                                                                  ###
########################################################################
########################################################################


########################################################################
##
##  condor_config
##
##  This is the global configuration file for condor.  Any settings
##  found here * * s h o u l d   b e   c u s t o m i z e d   i n
##  a   l o c a l   c o n f i g u r a t i o n   f i l e. * *
##
##  The local configuration files are located in LOCAL_CONFIG_DIR, set
##  below.
##
##  For a basic configuration, you may only want to start by
##  customizing CONDOR_HOST and DAEMON_LIST.
##
##  Note: To double-check where a configuration variable is set from
##  you can use condor_config_val -v -config <variable name>,
##  e.g. condor_config_val -v -config CONDOR_HOST.
##
##  The file is divided into four main parts:
##  Part 1:  Settings you likely want to customize
##  Part 2:  Settings you may want to customize
##  Part 3:  Settings that control the policy of when condor will
##           start and stop jobs on your machines
##  Part 4:  Settings you should probably leave alone (unless you
##  know what you're doing)
##
##  Please read the INSTALL file (or the Install chapter in the
##  Condor Administrator's Manual) for detailed explanations of the
##  various settings in here and possible ways to configure your
##  pool.
##
##  Unless otherwise specified, settings that are commented out show
##  the defaults that are used if you don't define a value.  Settings
##  that are defined here MUST BE DEFINED since they have no default
##  value.
##
##  Unless otherwise indicated, all settings which specify a time are
##  defined in seconds.
##
########################################################################

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

```
##  Part 1:  Settings you likely want to customize:
######################################################################
######################################################################

##  What machine is your central manager?
CONDOR_HOST = central-manager-hostname.your.domain

##--------------------------------------------------------------------
##  Pathnames:
##--------------------------------------------------------------------
##  Where have you installed the bin, sbin and lib condor directories?
RELEASE_DIR  = /usr

##  Where is the local condor directory for each host?
##  This is where the local config file(s), logs and
##  spool/execute directories are located
LOCAL_DIR  = $(TILDE)
#LOCAL_DIR  = $(RELEASE_DIR)/hosts/$(HOSTNAME)

## Looking for LOCAL_CONFIG_FILE? You will not find it here. Instead
## put a file in the LOCAL_CONFIG_DIR below. It is a more extensible
## means to manage configuration. The order in which configuration
## files are read from the LOCAL_CONFIG_DIR is lexicographic. For
## instance, config in 00MyConfig will be overridden by config in
## 97MyConfig.

##  Where are optional machine-specific local config files located?
##  Config files are included in lexicographic order.
##  No default.
LOCAL_CONFIG_DIR = $(ETC)/config.d

## If the local config file is not present, is it an error?
## WARNING: This is a potential security issue.
## If not specificed, the default is True
#REQUIRE_LOCAL_CONFIG_FILE = TRUE

##--------------------------------------------------------------------
##  Mail parameters:
##--------------------------------------------------------------------
##  When something goes wrong with condor at your site, who should get
##  the email?
CONDOR_ADMIN  = root@$(FULL_HOSTNAME)

##  Full path to a mail delivery program that understands that "-s"
##  means you want to specify a subject:
MAIL   = /bin/mail

##--------------------------------------------------------------------
##  Network domain parameters:
##--------------------------------------------------------------------
##  Internet domain of machines sharing a common UID space.  If your
##  machines don't share a common UID space, set it to
##  UID_DOMAIN = $(FULL_HOSTNAME)
##  to specify that each machine has its own UID space.
UID_DOMAIN  = $(FULL_HOSTNAME)

##  Internet domain of machines sharing a common file system.
##  If your machines don't use a network file system, set it to
##  FILESYSTEM_DOMAIN = $(FULL_HOSTNAME)
##  to specify that each machine has its own file system.
FILESYSTEM_DOMAIN = $(FULL_HOSTNAME)

##  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''.
COLLECTOR_NAME   = My Pool - $(CONDOR_HOST)

######################################################################
######################################################################
##
##  ######                                   #####
##  #     #    ##    #####    #####         #     #
##  #     #   #  #   #    #     #                 #
##  ######   #    #  #    #     #            #####
##  #        ######  #####      #           #
##  #        #    #  #   #      #           #
##  #        #    #  #    #     #           #######
##
##  Part 2:  Settings you may want to customize:
##  (it is generally safe to leave these untouched)
######################################################################
######################################################################

##
##  The user/group ID <uid>.<gid> of the "Condor" user.
##  (this can also be specified in the environment)
##  Note: the CONDOR_IDS setting is ignored on Win32 platforms
#CONDOR_IDS=x.x

##--------------------------------------------------------------------
##  Flocking: Submitting jobs to more than one pool
##--------------------------------------------------------------------
##  Flocking allows you to run your jobs in other pools, or lets
##  others run jobs in your pool.
##
##  To let others flock to you, define FLOCK_FROM.
##
##  To flock to others, define FLOCK_TO.

##  FLOCK_FROM defines the machines where you would like to grant
##  people access to your pool via flocking. (i.e. you are granting
##  access to these machines to join your pool).
FLOCK_FROM =
##  An example of this is:
#FLOCK_FROM = somehost.friendly.domain, anotherhost.friendly.domain

##  FLOCK_TO defines the central managers of the pools that you want
##  to flock to. (i.e. you are specifying the machines that you
##  want your jobs to be negotiated at -- thereby specifying the
##  pools they will run in.)
FLOCK_TO =
##  An example of this is:
#FLOCK_TO = central_manager.friendly.domain, condor.cs.wisc.edu

##  FLOCK_COLLECTOR_HOSTS should almost always be the same as
##  FLOCK_NEGOTIATOR_HOSTS (as shown below).  The only reason it would be
##  different is if the collector and negotiator in the pool that you are
##  flocking too are running on different machines (not recommended).
##  The collectors must be specified in the same corresponding order as
##  the FLOCK_NEGOTIATOR_HOSTS list.
FLOCK_NEGOTIATOR_HOSTS = $(FLOCK_TO)
FLOCK_COLLECTOR_HOSTS = $(FLOCK_TO)
## An example of having the negotiator and the collector on different
## machines is:
#FLOCK_NEGOTIATOR_HOSTS = condor.cs.wisc.edu, condor-negotiator.friendly.domain
#FLOCK_COLLECTOR_HOSTS =  condor.cs.wisc.edu, condor-collector.friendly.domain

##--------------------------------------------------------------------
```

```
##  Host/IP access levels
##--------------------------------------------------------------------
##  Please see the administrator's manual for details on these
##  settings, what they're for, and how to use them.

##  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).
ALLOW_ADMINISTRATOR = $(CONDOR_HOST)

##  If there are no machines that should have administrative access
##  to your pool (for example, there's no machine where only trusted
##  users have accounts), you can uncomment this setting.
##  Unfortunately, this will mean that administering your pool will
##  be more difficult.
#DENY_ADMINISTRATOR = *

##  What machines should have "owner" access to your machines, meaning
##  they can issue commands that a machine owner should be able to
##  issue to their own machine (like condor_vacate).  This defaults to
##  machines with administrator access, and the local machine.  This
##  is probably what you want.
ALLOW_OWNER = $(FULL_HOSTNAME), $(ALLOW_ADMINISTRATOR)

##  Read access.  Machines listed as allow (and/or not listed as deny)
##  can view the status of your pool, but cannot join your pool
##  or run jobs.
##  NOTE: By default, without these entries customized, you
##  are granting read access to the whole world.  You may want to
##  restrict that to hosts in your domain.  If possible, please also
##  grant read access to "*.cs.wisc.edu", so the Condor developers
##  will be able to view the status of your pool and more easily help
##  you install, configure or debug your Condor installation.
##  It is important to have this defined.
ALLOW_READ = *
#ALLOW_READ = *.your.domain, *.cs.wisc.edu
#DENY_READ = *.bad.subnet, bad-machine.your.domain, 144.77.88.*

##  Write access.  Machines listed here can join your pool, submit
##  jobs, etc.  Note: Any machine which has WRITE access must
##  also be granted READ access.  Granting WRITE access below does
##  not also automatically grant READ access; you must change
##  ALLOW_READ above as well.
##
##  You must set this to something else before Condor will run.
##  This most simple option is:
##    ALLOW_WRITE = *
##  but note that this will allow anyone to submit jobs or add
##  machines to your pool and is a serious security risk.

ALLOW_WRITE = $(FULL_HOSTNAME)
#ALLOW_WRITE = *.your.domain, your-friend's-machine.other.domain
#DENY_WRITE = bad-machine.your.domain

##  Are you upgrading to a new version of Condor and confused about
##  why the above ALLOW_WRITE setting is causing Condor to refuse to
##  start up?  If you are upgrading from a configuration that uses
##  HOSTALLOW/HOSTDENY instead of ALLOW/DENY we recommend that you
##  convert all uses of the former to the latter.  The syntax of the
##  authorization settings is identical.  They both support
##  unauthenticated IP-based authorization as well as authenticated
##  user-based authorization.  To avoid confusion, the use of
##  HOSTALLOW/HOSTDENY is discouraged.  Support for it may be removed
##  in the future.

##  Negotiator access.  Machines listed here are trusted central
##  managers.  You should normally not have to change this.
ALLOW_NEGOTIATOR = $(CONDOR_HOST)
##  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.
ALLOW_NEGOTIATOR_SCHEDD = $(CONDOR_HOST), $(FLOCK_NEGOTIATOR_HOSTS)

##  Config access.  Machines listed here can use the condor_config_val
##  tool to modify all daemon configurations.  This level of host-wide
##  access should only be granted with extreme caution.  By default,
##  config access is denied from all hosts.
#ALLOW_CONFIG = trusted-host.your.domain

##  Flocking Configs.  These are the real things that Condor looks at,
##  but we set them from the FLOCK_FROM/TO macros above.  It is safe
##  to leave these unchanged.
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)


##--------------------------------------------------------------------
##  Security parameters for setting configuration values remotely:
##--------------------------------------------------------------------
##  These parameters define the list of attributes that can be set
##  remotely with condor_config_val for the security access levels
##  defined above (for example, WRITE, ADMINISTRATOR, CONFIG, etc).
##  Please see the administrator's manual for futher details on these
##  settings, what they're for, and how to use them.  There are no
##  default values for any of these settings.  If they are not
##  defined, no attributes can be set with condor_config_val.

## Do you want to allow condor_config_val -rset to work at all?
## This feature is disabled by default, so to enable, you must
## uncomment the following setting and change the value to "True".
## Note: changing this requires a restart not just a reconfig.
#ENABLE_RUNTIME_CONFIG = False

## Do you want to allow condor_config_val -set to work at all?
## This feature is disabled by default, so to enable, you must
## uncomment the following setting and change the value to "True".
## Note: changing this requires a restart not just a reconfig.
#ENABLE_PERSISTENT_CONFIG = False

## Directory where daemons should write persistent config 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 must define this.
## Note: changing this requires a restart not just a reconfig.
#PERSISTENT_CONFIG_DIR = /full/path/to/root-only/local/directory

##  Attributes that can be set by hosts with "CONFIG" permission (as
##  defined with ALLOW_CONFIG and DENY_CONFIG above).
##  The commented-out value here was the default behavior of Condor
##  prior to version 6.3.3.  If you don't need this behavior, you
##  should leave this commented out.
#SETTABLE_ATTRS_CONFIG = *

##  Attributes that can be set by hosts with "ADMINISTRATOR"
##  permission (as defined above)
#SETTABLE_ATTRS_ADMINISTRATOR = *_DEBUG, MAX_*_LOG

##  Attributes that can be set by hosts with "OWNER" permission (as
##  defined above) NOTE: any Condor job running on a given host will
##  have OWNER permission on that host by default.  If you grant this
##  kind of access, Condor jobs will be able to modify any attributes
##  you list below on the machine where they are running.  This has
##  obvious security implications, so only grant this kind of
##  permission for custom attributes that you define for your own use
##  at your pool (custom attributes about your machines that are
##  published with the STARTD_ATTRS setting, for example).
#SETTABLE_ATTRS_OWNER = your_custom_attribute, another_custom_attr

##  You can also define daemon-specific versions of each of these
##  settings.  For example, to define settings that can only be
##  changed in the condor_startd's configuration by hosts with OWNER
##  permission, you would use:
#STARTD_SETTABLE_ATTRS_OWNER = your_custom_attribute_name


##--------------------------------------------------------------------
##  Network filesystem parameters:
##--------------------------------------------------------------------
##  Do you want to use NFS for file access instead of remote system
##  calls?
#USE_NFS  = False

##  Do you want to use AFS for file access instead of remote system
##  calls?
#USE_AFS  = False

##--------------------------------------------------------------------
##  Checkpoint server:
##--------------------------------------------------------------------
##  Do you want to use a checkpoint server if one is available?  If a
##  checkpoint server isn't available or USE_CKPT_SERVER is set to
##  False, checkpoints will be written to the local SPOOL directory on
##  the submission machine.
#USE_CKPT_SERVER = True

##  What's the hostname of this machine's nearest checkpoint server?
#CKPT_SERVER_HOST = checkpoint-server-hostname.your.domain

##  Do you want the starter on the execute machine to choose the
##  checkpoint server?  If False, the CKPT_SERVER_HOST set on
##  the submit machine is used.  Otherwise, the CKPT_SERVER_HOST set
##  on the execute machine is used.  The default is true.
#STARTER_CHOOSES_CKPT_SERVER = True

##--------------------------------------------------------------------
##  Miscellaneous:
##--------------------------------------------------------------------
##  Try to save this much swap space by not starting new shadows.
##  Specified in megabytes.
#RESERVED_SWAP  = 0

##  What's the maximum number of jobs you want a single submit machine
##  to spawn shadows for?  The default is a function of $(DETECTED_MEMORY)
##  and a guess at the number of ephemeral ports available.

## 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) )


##  Maximum number of simultaneous downloads of output files from
##  execute machines to the submit machine (limit applied per schedd).
##  The value 0 means unlimited.
#MAX_CONCURRENT_DOWNLOADS = 10

##  Maximum number of simultaneous uploads of input files from the
##  submit machine to execute machines (limit applied per schedd).
##  The value 0 means unlimited.
#MAX_CONCURRENT_UPLOADS = 10

##  Condor needs to create a few lock files to synchronize access to
##  various log files.  Because of problems we've had with network
##  filesystems and file locking over the years, we HIGHLY recommend
##  that you put these lock files on a local partition on each
##  machine.  If you don't 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're not running as root, this is whatever user you started up
##  the condor_master as.  If you are running as root, and there's a
##  condor account, it's probably condor.  Otherwise, it's whatever
##  you've set in the CONDOR_IDS environment variable.  See the Admin
##  manual for details on this.
LOCK  = /var/lock/condor

##  If you don't use a fully qualified name in your /etc/hosts file
##  (or NIS, etc.) for either your official hostname or as an alias,
##  Condor wouldn't normally be able to use fully qualified names in
##  places that it'd like to.  You can set this parameter to the
##  domain you'd like appended to your hostname, if changing your host
##  information isn't a good option.  This parameter must be set in
##  the global config file (not the LOCAL_CONFIG_FILE from above).
#DEFAULT_DOMAIN_NAME = your.domain.name

##  If you don't have DNS set up, Condor will normally fail in many
##  places because it can't resolve hostnames to IP addresses and
##  vice-versa. If you enable this option, Condor will use
##  pseudo-hostnames constructed from a machine's IP address and the
##  DEFAULT_DOMAIN_NAME. Both NO_DNS and DEFAULT_DOMAIN must be set in
##  your top-level config file for this mode of operation to work
##  properly.
#NO_DNS = True

##  Condor can be told whether or not you want the Condor daemons to
##  create a core file if something really bad happens.  This just
##  sets the resource limit for the size of a core file.  By default,
##  we don't do anything, and leave in place whatever limit was in
##  effect when you started the Condor daemons.  If this parameter is
##  set and "True", we increase the limit to as large as it gets.  If
##  it's set to "False", we set the limit at 0 (which means that no
##  core files are even created).  Core files greatly help the Condor
##  developers debug any problems you might be having.
#CREATE_CORE_FILES = True

##  When Condor daemons 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.
#ABORT_ON_EXCEPTION = False

##  Condor Glidein downloads binaries from a remote server for the
##  machines into which you're gliding. This saves you from manually
##  downloading and installing binaries for every architecture you
##  might want to glidein to. The default server is one maintained at
##  The University of Wisconsin. If you don't want to use the UW
##  server, you can set up your own and change the following to
##  point to it, instead.
GLIDEIN_SERVER_URLS = \
  http://www.cs.wisc.edu/condor/glidein/binaries

## List the sites you want to GlideIn to on the GLIDEIN_SITES. For example,
## if you'd like to GlideIn to some Alliance GiB resources,
## uncomment the line below.
## Make sure that $(GLIDEIN_SITES) is included in ALLOW_READ and
## ALLOW_WRITE, or else your GlideIns won't be able to join your pool.
## This is _NOT_ done for you by default, because it is an even better
## idea to use a strong security method (such as GSI) rather than
## host-based security for authorizing glideins.
#GLIDEIN_SITES = *.ncsa.uiuc.edu, *.cs.wisc.edu, *.mcs.anl.gov
#GLIDEIN_SITES =

##  If your site needs to use UID_DOMAIN settings (defined above) that
##  are not real Internet domains that match the hostnames, you can
##  tell Condor to trust whatever UID_DOMAIN a submit machine gives to
##  the execute machine and just make sure the two strings match.  The
##  default for this setting is False, since it is more secure this
##  way.
##   Default is False
TRUST_UID_DOMAIN = True

## If you would like to be informed in near real-time via condor_q when
## a vanilla/standard/java job is in a suspension state, set this attribute to
## TRUE. However, this real-time update of the condor_schedd by the shadows
## could cause performance issues if there are thousands of concurrently
## running vanilla/standard/java jobs under a single condor_schedd and they
## are allowed to suspend and resume.
#REAL_TIME_JOB_SUSPEND_UPDATES = False

## A standard universe job can perform arbitrary shell calls via the
## libc 'system()' function. This function call is routed back to the shadow
## which performs the actual system() invocation in the initialdir of the
## running program and as the user who submitted the job. However, since the
## user job can request ARBITRARY shell commands to be run by the shadow, this
## is a generally unsafe practice. This should only be made available if it is
## actually needed. If this attribute is not defined, then it is the same as
## it being defined to False. Set it to True to allow the shadow to execute
## arbitrary shell code from the user job.
#SHADOW_ALLOW_UNSAFE_REMOTE_EXEC = False

## KEEP_OUTPUT_SANDBOX is an optional feature to tell Condor-G to not
## remove the job spool when the job leaves the queue.  To use, just
## set to TRUE.  Since you will be operating Condor-G in this manner,
## you may want to put leave_in_queue = false in your job submit
## description files, to tell Condor-G to simply remove the job from
## the queue immediately when the job completes (since the output files
## will stick around no matter what).
#KEEP_OUTPUT_SANDBOX = False

## This setting tells the negotiator to ignore user priorities.  This
## avoids problems where jobs from different users won't run when using
## condor_advertise instead of a full-blown startd (some of the user
## priority system in Condor relies on information from the startd --
## we will remove this reliance when we support the user priority
## system for grid sites in the negotiator; for now, this setting will
## just disable it).
#NEGOTIATOR_IGNORE_USER_PRIORITIES = False

## These are the directories used to locate classad plug-in functions
#CLASSAD_SCRIPT_DIRECTORY =
#CLASSAD_LIB_PATH =

## This setting tells Condor whether to delegate or copy GSI X509
## credentials when sending them over the wire between daemons.
## Delegation can take up to a second, which is very slow when
## submitting a large number of jobs. Copying exposes the credential
## to third parties if Condor isn't set to encrypt communications.
## By default, Condor will delegate rather than copy.
#DELEGATE_JOB_GSI_CREDENTIALS = True

## This setting controls whether Condor delegates a full or limited
## X509 credential for jobs. Currently, this only affects grid-type
## gt2 grid universe jobs. The default is False.
#DELEGATE_FULL_JOB_GSI_CREDENTIALS = False

## This setting controls the default behaviour for the spooling of files
## into, or out of, the Condor system by such tools as condor_submit
## and condor_transfer_data. Here is the list of valid settings for this
## parameter and what they mean:
##
##   stm_use_schedd_only
##      Ask the condor_schedd to solely store/retreive the sandbox
##
##   stm_use_transferd
##      Ask the condor_schedd for a location of a condor_transferd, then
##      store/retreive the sandbox from the transferd itself.
##
## The allowed values are case insensitive.
## The default of this parameter if not specified is: stm_use_schedd_only
#SANDBOX_TRANSFER_METHOD = stm_use_schedd_only

##--------------------------------------------------------------------
##  Settings that control the daemon's debugging output:
##--------------------------------------------------------------------

##
## The flags given in ALL_DEBUG are shared between all daemons.
```

```
##

ALL_DEBUG               =

MAX_COLLECTOR_LOG = 1000000
COLLECTOR_DEBUG  =

MAX_KBDD_LOG  = 1000000
KBDD_DEBUG  =

MAX_NEGOTIATOR_LOG = 1000000
NEGOTIATOR_DEBUG = D_MATCH
MAX_NEGOTIATOR_MATCH_LOG = 1000000

MAX_SCHEDD_LOG  = 1000000
SCHEDD_DEBUG  = D_PID

MAX_SHADOW_LOG  = 1000000
SHADOW_DEBUG  =

MAX_STARTD_LOG  = 1000000
STARTD_DEBUG  =

MAX_STARTER_LOG  = 1000000

MAX_MASTER_LOG  = 1000000
MASTER_DEBUG  =
##  When the master starts up, should it truncate it's log file?
#TRUNC_MASTER_LOG_ON_OPEN        = False

MAX_JOB_ROUTER_LOG      = 1000000
JOB_ROUTER_DEBUG        =

MAX_ROOSTER_LOG         = 1000000
ROOSTER_DEBUG           =

MAX_HDFS_LOG            = 1000000
HDFS_DEBUG              =

MAX_TRIGGERD_LOG      = 1000000
TRIGGERD_DEBUG        =

# High Availability Logs
MAX_HAD_LOG  = 1000000
HAD_DEBUG  =
MAX_REPLICATION_LOG = 1000000
REPLICATION_DEBUG =
MAX_TRANSFERER_LOG = 1000000
TRANSFERER_DEBUG =


## The daemons touch their log file periodically, even when they have
## nothing to write. When a daemon starts up, it prints the last time
## the log file was modified. This lets you estimate when a previous
## instance of a daemon stopped running. This paramete controls how often
## the daemons touch the file (in seconds).
#TOUCH_LOG_INTERVAL = 60

######################################################################
######################################################################
##
##  ######                                   #####
##  #     #    ##    #####    #####         #     #
##  #     #   #  #   #    #     #                 #
##  ######   #    #  #    #     #            #####
##  #        ######  #####      #                #
##  #        #    #  #   #      #           #    #
##  #        #    #  #    #     #            #####
##
##  Part 3:  Settings control the policy for running, stopping, and
##  periodically checkpointing condor jobs:
######################################################################
######################################################################

##  This section contains macros are here to help write legible
##  expressions:
MINUTE  = 60
HOUR  = (60 * $(MINUTE))
StateTimer = (CurrentTime - EnteredCurrentState)
ActivityTimer = (CurrentTime - EnteredCurrentActivity)
ActivationTimer = ifThenElse(JobStart =!= UNDEFINED, (CurrentTime - JobStart), 0)
LastCkpt = (CurrentTime - LastPeriodicCheckpoint)

##  The JobUniverse attribute is just an int.  These macros can be
##  used to specify the universe in a human-readable way:
STANDARD = 1
VANILLA  = 5
MPI  = 8
VM  = 13
IsMPI           = (TARGET.JobUniverse == $(MPI))
IsVanilla       = (TARGET.JobUniverse == $(VANILLA))
IsStandard      = (TARGET.JobUniverse == $(STANDARD))
IsVM            = (TARGET.JobUniverse == $(VM))

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)

BigJob  = (TARGET.ImageSize >= (50 * 1024))
MediumJob = (TARGET.ImageSize >= (15 * 1024) && TARGET.ImageSize < (50 * 1024))
SmallJob = (TARGET.ImageSize <  (15 * 1024))

JustCPU   = ($(CPUBusy) && ($(KeyboardBusy) == False))
MachineBusy  = ($(CPUBusy) || $(KeyboardBusy))

##  The RANK expression controls which jobs this machine prefers to
##  run over others.  Some examples from the manual include:
##    RANK = TARGET.ImageSize
##    RANK = (Owner == "coltrane") + (Owner == "tyner") \
##                   + ((Owner == "garrison") * 10) + (Owner == "jones")
##  By default, RANK is always 0, meaning that all jobs have an equal
##  ranking.
#RANK   = 0


#####################################################################
##  This where you choose the configuration that you would like to
##  use.  It has no defaults so it must be defined.  We start this
##  file off with the UWCS_* policy.
######################################################################

##  Also here is what is referred to as the TESTINGMODE_*, which is
##  a quick hardwired way to test Condor with a simple no-preemption policy.
##  Replace UWCS_* with TESTINGMODE_* if you wish to do testing mode.
##  For example:
##  WANT_SUSPEND   = $(UWCS_WANT_SUSPEND)
##  becomes
##  WANT_SUSPEND   = $(TESTINGMODE_WANT_SUSPEND)

# When should we only consider SUSPEND instead of PREEMPT?
WANT_SUSPEND   = $(UWCS_WANT_SUSPEND)

# When should we preempt gracefully instead of hard-killing?
WANT_VACATE  = $(UWCS_WANT_VACATE)

##  When is this machine willing to start a job?
START   = $(UWCS_START)

##  When should a local universe job be allowed to start?
#START_LOCAL_UNIVERSE = TotalLocalJobsRunning < 200

##  When should a scheduler universe job be allowed to start?
#START_SCHEDULER_UNIVERSE = TotalSchedulerJobsRunning < 200

##  When to suspend a job?
SUSPEND   = $(UWCS_SUSPEND)

##  When to resume a suspended job?
CONTINUE  = $(UWCS_CONTINUE)

##  When to nicely stop a job?
##  (as opposed to killing it instantaneously)
PREEMPT   = $(UWCS_PREEMPT)

##  When to instantaneously kill a preempting job
##  (e.g. if a job is in the pre-empting stage for too long)
KILL   = $(UWCS_KILL)

PERIODIC_CHECKPOINT = $(UWCS_PERIODIC_CHECKPOINT)
PREEMPTION_REQUIREMENTS = $(UWCS_PREEMPTION_REQUIREMENTS)
PREEMPTION_RANK  = $(UWCS_PREEMPTION_RANK)
NEGOTIATOR_PRE_JOB_RANK = $(UWCS_NEGOTIATOR_PRE_JOB_RANK)
NEGOTIATOR_POST_JOB_RANK = $(UWCS_NEGOTIATOR_POST_JOB_RANK)
MaxJobRetirementTime    = $(UWCS_MaxJobRetirementTime)
CLAIM_WORKLIFE          = $(UWCS_CLAIM_WORKLIFE)

#####################################################################
## This is the UWisc - CS Department Configuration.
#####################################################################

# When should we only consider SUSPEND instead of PREEMPT?
# Only when SUSPEND is True and one of the following is also true:
#   - the job is small
#   - the keyboard is idle
#   - it is a vanilla universe job
UWCS_WANT_SUSPEND  = ( $(SmallJob) || $(KeyboardNotBusy) || $(IsVanilla) ) && \
                     ( $(SUSPEND) )

# When should we preempt gracefully instead of hard-killing?
UWCS_WANT_VACATE   = ( $(ActivationTimer) > 10 * $(MINUTE) || $(IsVanilla) )

# Only start jobs if:
```

```
# 1) the keyboard has been idle long enough, AND
# 2) the load average is low enough OR the machine is currently
#    running a Condor job
# (NOTE: Condor will only run 1 job at a time on a given resource.
# The reasons Condor might consider running a different job while
# already running one are machine Rank (defined above), and user
# priorities.)
UWCS_START = ( (KeyboardIdle > $(StartIdleTime)) \
                    && ( $(CPUIdle) || \
                         (State != "Unclaimed" && State != "Owner")) )

# Suspend jobs if:
# 1) the keyboard has been touched, OR
# 2a) The cpu has been busy for more than 2 minutes, AND
# 2b) the job has been running for more than 90 seconds
UWCS_SUSPEND = ( $(KeyboardBusy) || \
                 ( (CpuBusyTime > 2 * $(MINUTE)) \
                   && $(ActivationTimer) > 90 ) )

# Continue jobs if:
# 1) the cpu is idle, AND
# 2) we've been suspended more than 10 seconds, AND
# 3) the keyboard hasn't been touched in a while
UWCS_CONTINUE = ( $(CPUIdle) && ($(ActivityTimer) > 10) \
                  && (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)
UWCS_PREEMPT = ( ((Activity == "Suspended") && \
                  ($(ActivityTimer) > $(MaxSuspendTime))) \
   || (SUSPEND && (WANT_SUSPEND == False)) )

# Maximum time (in seconds) to wait for a job to finish before kicking
# it off (due to PREEMPT, a higher priority claim, or the startd
# gracefully shutting down).  This is computed from the time the job
# was started, minus any suspension time.  Once the retirement time runs
# out, the usual preemption process will take place.  The job may
# self-limit the retirement time to _less_ than what is given here.
# By default, nice user jobs and standard universe jobs set their
# MaxJobRetirementTime to 0, so they will not wait in retirement.

UWCS_MaxJobRetirementTime = 0

##  If you completely disable preemption of claims to machines, you
##  should consider limiting the timespan over which new jobs will be
##  accepted on the same claim.  See the manual section on disabling
##  preemption for a comprehensive discussion.  Since this example
##  configuration does not disable preemption of claims, we leave
##  CLAIM_WORKLIFE undefined (infinite).
#UWCS_CLAIM_WORKLIFE = 1200

# Kill jobs if they have taken too long to vacate gracefully
UWCS_KILL = $(ActivityTimer) > $(MaxVacateTime)

##  Only define vanilla versions of these if you want to make them
##  different from the above settings.
#SUSPEND_VANILLA  = ( $(KeyboardBusy) || \
#       ((CpuBusyTime > 2 * $(MINUTE)) && $(ActivationTimer) > 90) )
#CONTINUE_VANILLA = ( $(CPUIdle) && ($(ActivityTimer) > 10) \
#                     && (KeyboardIdle > $(ContinueIdleTime)) )
#PREEMPT_VANILLA  = ( ((Activity == "Suspended") && \
#                     ($(ActivityTimer) > $(MaxSuspendTime))) \
#                      || (SUSPEND_VANILLA && (WANT_SUSPEND == False)) )
#KILL_VANILLA    = $(ActivityTimer) > $(MaxVacateTime)

##  Checkpoint every 3 hours on average, with a +-30 minute random
##  factor to avoid having many jobs hit the checkpoint server at
##  the same time.
UWCS_PERIODIC_CHECKPOINT = $(LastCkpt) > (3 * $(HOUR) + \
                                  $RANDOM_INTEGER(-30,30,1) * $(MINUTE) )

##  You might want to checkpoint a little less often.  A good
##  example of this is below.  For jobs smaller than 60 megabytes, we
##  periodic checkpoint every 6 hours.  For larger jobs, we only
##  checkpoint every 12 hours.
#UWCS_PERIODIC_CHECKPOINT = \
#          ( (TARGET.ImageSize < 60000) && \
#            ($(LastCkpt) > (6  * $(HOUR) + $RANDOM_INTEGER(-30,30,1))) ) || \
#          (  $(LastCkpt) > (12 * $(HOUR) + $RANDOM_INTEGER(-30,30,1)) )

##  The rank expressions used by the negotiator are configured below.
##  This is the order in which ranks are applied by the negotiator:
##    1. NEGOTIATOR_PRE_JOB_RANK
##    2. rank in job ClassAd
##    3. NEGOTIATOR_POST_JOB_RANK
##    4. cause of preemption (0=user priority,1=startd rank,2=no preemption)
##    5. PREEMPTION_RANK

##  The NEGOTIATOR_PRE_JOB_RANK expression overrides all other ranks
##  that are used to pick a match from the set of possibilities.
##  The following expression matches jobs to unclaimed resources
##  whenever possible, regardless of the job-supplied rank.
UWCS_NEGOTIATOR_PRE_JOB_RANK = RemoteOwner =?= UNDEFINED

##  The NEGOTIATOR_POST_JOB_RANK expression chooses between
##  resources that are equally preferred by the job.
##  The following example expression steers jobs toward
##  faster machines and tends to fill a cluster of multi-processors
##  breadth-first instead of depth-first.  It also prefers online
##  machines over offline (hibernating) ones.  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 - 1.0e10*(Offline=?=True))

##  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.  This expression defaults to true.
UWCS_PREEMPTION_REQUIREMENTS = ( $(StateTimer) > (1 * $(HOUR)) && \
 RemoteUserPrio > SubmitterUserPrio * 1.2 ) || (MY.NiceUser == True)

##  The PREEMPTION_RANK expression is used in a case where preemption
##  is the only option and all other negotiation ranks are equal.  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 all other
##  negotiation ranks are equal.
UWCS_PREEMPTION_RANK = (RemoteUserPrio * 1000000) - TARGET.ImageSize


#####################################################################
##  This is a Configuration that will cause your Condor jobs to
##  always run.  This is intended for testing only.
######################################################################

##  This mode will cause your jobs to start on a machine an will let
##  them run to completion.  Condor will ignore all of what is going
##  on in the machine (load average, keyboard activity, etc.)

TESTINGMODE_WANT_SUSPEND = False
TESTINGMODE_WANT_VACATE  = False
TESTINGMODE_START   = True
TESTINGMODE_SUSPEND   = False
TESTINGMODE_CONTINUE  = True
TESTINGMODE_PREEMPT   = False
TESTINGMODE_KILL   = False
TESTINGMODE_PERIODIC_CHECKPOINT = False
TESTINGMODE_PREEMPTION_REQUIREMENTS = False
TESTINGMODE_PREEMPTION_RANK = 0

# Prevent machine claims from being reused indefinitely, since
# preemption of claims is disabled in the TESTINGMODE configuration.
TESTINGMODE_CLAIM_WORKLIFE = 1200


######################################################################
######################################################################
##
##  ######                                  #
##  #     #    ##    #####    #####         #    #
##  #     #   #  #   #    #     #           #    #
##  ######   #    #  #    #     #           #    #
##  #        ######  #####      #           #######
##  #        #    #  #   #      #                #
##  #        #    #  #    #     #                #
##
##  Part 4:  Settings you should probably leave alone:
##  (unless you know what you're doing)
######################################################################
######################################################################

######################################################################
##  Daemon-wide settings:
######################################################################

##  Pathnames
LOG  = /var/log/condor
SPOOL  = $(LOCAL_DIR)/spool
EXECUTE  = $(LOCAL_DIR)/execute
BIN  = $(RELEASE_DIR)/bin
LIB = $(RELEASE_DIR)/lib64/condor
INCLUDE  = $(RELEASE_DIR)/include/condor
SBIN  = $(RELEASE_DIR)/sbin
LIBEXEC  = $(RELEASE_DIR)/libexec/condor
SHARE  = $(RELEASE_DIR)/share/condor
RUN  = /var/run/condor
DATA            = $(SPOOL)
ETC  = /etc/condor

## If you leave HISTORY undefined (comment it out), no history file
## will be created.
HISTORY  = $(SPOOL)/history

##  Log files
COLLECTOR_LOG = $(LOG)/CollectorLog
KBDD_LOG = $(LOG)/KbdLog
```

```
MASTER_LOG = $(LOG)/MasterLog
NEGOTIATOR_LOG = $(LOG)/NegotiatorLog
NEGOTIATOR_MATCH_LOG = $(LOG)/MatchLog
SCHEDD_LOG = $(LOG)/SchedLog
SHADOW_LOG = $(LOG)/ShadowLog
STARTD_LOG = $(LOG)/StartLog
STARTER_LOG = $(LOG)/StarterLog
JOB_ROUTER_LOG  = $(LOG)/JobRouterLog
ROOSTER_LOG     = $(LOG)/RoosterLog
SHARED_PORT_LOG = $(LOG)/SharedPortLog
TRIGGERD_LOG  = $(LOG)/TriggerLog

# High Availability Logs
HAD_LOG  = $(LOG)/HADLog
REPLICATION_LOG = $(LOG)/ReplicationLog
TRANSFERER_LOG = $(LOG)/TransfererLog
HDFS_LOG = $(LOG)/HDFSLog

##  Lock files
SHADOW_LOCK = $(LOCK)/ShadowLock

## This setting controls how often any lock files currently in use have their
## timestamp updated. Updating the timestamp prevents administrative programs
## like 'tmpwatch' from deleting long lived lock files. The parameter is
## an integer in seconds with a minimum of 60 seconds. The default if not
## specified is 28800 seconds, or 8 hours.
## This attribute only takes effect on restart of the daemons or at the next
## update time.
# LOCK_FILE_UPDATE_INTERVAL = 28800

##  This setting primarily allows you to change the port that the
##  collector is listening on.  By default, the collector uses port
##  9618, but you can set the port with a ":port", such as:
##  COLLECTOR_HOST = $(CONDOR_HOST):1234
COLLECTOR_HOST  = $(CONDOR_HOST)

## The NEGOTIATOR_HOST parameter has been deprecated.  The port where
## the negotiator is listening is now dynamically allocated and the IP
## and port are now obtained from the collector, just like all the
## other daemons.  However, if your pool contains any machines that
## are running version 6.7.3 or earlier, you can uncomment this
## setting to go back to the old fixed-port (9614) for the negotiator.
#NEGOTIATOR_HOST = $(CONDOR_HOST)

##  How long are you willing to let daemons try their graceful
##  shutdown methods before they do a hard shutdown? (30 minutes)
#SHUTDOWN_GRACEFUL_TIMEOUT = 1800

##  How much disk space would you like reserved from Condor?  In
##  places where Condor is computing the free disk space on various
##  partitions, it subtracts the amount it really finds by this
##  many megabytes.  (If undefined, defaults to 0).
RESERVED_DISK  = 5

##  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 to true.
#RESERVE_AFS_CACHE = False

##  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 hostname 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.  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 won't work.
#EMAIL_DOMAIN = $(FULL_HOSTNAME)

##  Should Condor daemons create a UDP command socket (for incomming
##  UDP-based commands) in addition to the TCP command socket?  By
##  default, classified ad updates sent to the collector use UDP, in
##  addition to some 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).  If not
##  defined, the UDP command socket is enabled by default, and to
##  modify this, you must restart your Condor daemons. Also, this
##  setting must be defined machine-wide.  For example, setting
##  "STARTD.WANT_UDP_COMMAND_SOCKET = False" while the global setting
##  is "True" will still result in the startd creating a UDP socket.
#WANT_UDP_COMMAND_SOCKET = True

##  If your site needs to use TCP updates to the collector, instead of
##  UDP, you can enable this feature.  HOWEVER, WE DO NOT RECOMMEND
##  THIS FOR MOST SITES!  In general, the only sites that might want
##  this feature are pools made up of machines connected via a
##  wide-area network where UDP packets are frequently or always
##  dropped.  If you enable this feature, you *MUST* turn on the
##  COLLECTOR_SOCKET_CACHE_SIZE setting at your collector, and each
##  entry in the socket cache uses another file descriptor.  If not
##  defined, this feature is disabled by default.
#UPDATE_COLLECTOR_WITH_TCP = True

## HIGHPORT and LOWPORT let you set the range of ports that Condor
## will use. This may be useful if you are behind a firewall. By
## default, Condor uses port 9618 for the collector, 9614 for the
## negotiator, and system-assigned (apparently random) ports for
## everything else. HIGHPORT and LOWPORT only affect these
## system-assigned ports, but will restrict them to the range you
## specify here. If you want to change the well-known ports for the
## collector or negotiator, see COLLECTOR_HOST or NEGOTIATOR_HOST.
## Note that both LOWPORT and HIGHPORT must be at least 1024 if you
## are not starting your daemons as root.  You may also specify
## different port ranges for incoming and outgoing connections by
## using IN_HIGHPORT/IN_LOWPORT and OUT_HIGHPORT/OUT_LOWPORT.
#HIGHPORT = 9700
#LOWPORT = 9600

##  If a daemon doens't respond for too long, do you want go generate
##  a core file?  This bascially controls the type of the signal
##  sent to the child process, and mostly affects the Condor Master
#NOT_RESPONDING_WANT_CORE = False


######################################################################
##  Daemon-specific settings:
######################################################################

##--------------------------------------------------------------------
##  condor_master
```

```
##--------------------------------------------------------------------
##  Daemons you want the master to keep running for you:
DAEMON_LIST   = MASTER, STARTD, SCHEDD

##  Which daemons use the Condor DaemonCore library (i.e., not the
##  checkpoint server or custom user daemons)?
#DC_DAEMON_LIST = \
#MASTER, STARTD, SCHEDD, KBDD, COLLECTOR, NEGOTIATOR, EVENTD, \
#VIEW_SERVER, CONDOR_VIEW, VIEW_COLLECTOR, HAWKEYE, CREDD, HAD, \
#DBMSD, QUILL, JOB_ROUTER, ROOSTER, LEASEMANAGER, HDFS, SHARED_PORT, TRIGGERD


##  Where are the binaries for these daemons?
MASTER    = $(SBIN)/condor_master
STARTD    = $(SBIN)/condor_startd
SCHEDD    = $(SBIN)/condor_schedd
KBDD    = $(SBIN)/condor_kbdd
NEGOTIATOR   = $(SBIN)/condor_negotiator
COLLECTOR   = $(SBIN)/condor_collector
STARTER_LOCAL   = $(SBIN)/condor_starter
JOB_ROUTER                      = $(LIBEXEC)/condor_job_router
ROOSTER                         = $(LIBEXEC)/condor_rooster
HDFS    = $(LIBEXEC)/condor_hdfs
SHARED_PORT   = $(LIBEXEC)/condor_shared_port
TRIGGERD                        = $(SBIN)/condor_triggerd

##  When the master starts up, it can place it's address (IP and port)
##  into a file.  This way, tools running on the local machine don't
##  need to query the central manager to find the master.  This
##  feature can be turned off by commenting out this setting.
MASTER_ADDRESS_FILE = $(LOG)/.master_address

##  Where should the master find the condor_preen binary? If you don't
##  want preen to run at all, just comment out this setting.
PREEN    = $(SBIN)/condor_preen

##  How do you want preen to behave?  The "-m" means you want email
##  about files preen finds that it thinks it should remove.  The "-r"
##  means you want preen to actually remove these files.  If you don't
##  want either of those things to happen, just remove the appropriate
##  one from this setting.
PREEN_ARGS   = -m -r

##  How often should the master start up condor_preen? (once a day)
#PREEN_INTERVAL   = 86400

##  If a daemon dies an unnatural death, do you want email about it?
#PUBLISH_OBITUARIES  = True

##  If you're getting obituaries, how many lines of the end of that
##  daemon's log file do you want included in the obituary?
#OBITUARY_LOG_LENGTH  = 20

##  Should the master run?
#START_MASTER   = True

##  Should the master start up the daemons you want it to?
#START_DAEMONS   = True

##  How often do you want the master to send an update to the central
##  manager?
#MASTER_UPDATE_INTERVAL  = 300

##  How often do you want the master to check the timestamps of the
##  daemons it's running?  If any daemons have been modified, the
##  master restarts them.
#MASTER_CHECK_NEW_EXEC_INTERVAL = 300

##  Once you notice new binaries, how long should you wait before you
##  try to execute them?
#MASTER_NEW_BINARY_DELAY = 120

##  What's the maximum amount of time you're willing to give the
##  daemons to quickly shutdown before you just kill them outright?
#SHUTDOWN_FAST_TIMEOUT  = 120

######
##  Exponential backoff settings:
######
##  When a daemon keeps crashing, we use "exponential backoff" so we
##  wait longer and longer before restarting it.  This is the base of
##  the exponent used to determine how long to wait before starting
##  the daemon again:
#MASTER_BACKOFF_FACTOR  = 2.0

##  What's 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, you'd hit 1 hour in 12 restarts...)
#MASTER_BACKOFF_CEILING  = 3600

##  How long should a daemon run without crashing before we consider
##  it "recovered".  Once a daemon has recovered, we reset the number
##  of restarts so the exponential backoff stuff goes back to normal.
#MASTER_RECOVER_FACTOR  = 300


##--------------------------------------------------------------------
##  condor_collector
##--------------------------------------------------------------------
## Address to which Condor will send a weekly e-mail with output of
## condor_status.
##  Default is condor-admin@cs.wisc.edu
CONDOR_DEVELOPERS = NONE

## Global Collector to periodically advertise basic information about
## your pool.
##  Default is condor.cs.wisc.edu
CONDOR_DEVELOPERS_COLLECTOR = NONE


##--------------------------------------------------------------------
##  condor_negotiator
##--------------------------------------------------------------------
## Determine if the Negotiator will honor SlotWeight attributes, which
## may be used to give a slot greater weight when calculating usage.
##  Default: True
NEGOTIATOR_USE_SLOT_WEIGHTS = False

## How often the Negotaitor starts a negotiation cycle, defined in
## seconds.
#NEGOTIATOR_INTERVAL = 60


##--------------------------------------------------------------------
##  condor_startd
##--------------------------------------------------------------------
##  Where are the various condor_starter binaries installed?
STARTER_LIST = STARTER, STARTER_STANDARD
```

```
STARTER   = $(SBIN)/condor_starter
STARTER_STANDARD = $(SBIN)/condor_starter.std
STARTER_LOCAL  = $(SBIN)/condor_starter

##  When the startd starts up, it can place it's address (IP and port)
##  into a file.  This way, tools running on the local machine don't
##  need to query the central manager to find the startd.  This
##  feature can be turned off by commenting out this setting.
STARTD_ADDRESS_FILE = $(LOG)/.startd_address

##  When a machine is claimed, how often should we poll the state of
##  the machine to see if we need to evict/suspend the job, etc?
#POLLING_INTERVAL        = 5

##  How often should the startd send updates to the central manager?
#UPDATE_INTERVAL         = 300

##  How long is the startd willing to stay in the "matched" state?
#MATCH_TIMEOUT  = 300

##  How long is the startd willing to stay in the preempting/killing
##  state before it just kills the starter directly?
#KILLING_TIMEOUT = 30

##  When a machine unclaimed, when should it run benchmarks?
##  LastBenchmark is initialized to 0, so this expression says as soon
##  as we're unclaimed, run the benchmarks.  Thereafter, if we're
##  unclaimed and it's been at least 4 hours since we ran the last
##  benchmarks, run them again.  The startd keeps a weighted average
##  of the benchmark results to provide more accurate values.
##  Note, if you don't want any benchmarks run at all, either comment
##  RunBenchmarks out, or set it to "False".
BenchmarkTimer = (CurrentTime - LastBenchmark)
RunBenchmarks : (LastBenchmark == 0 ) || ($(BenchmarkTimer) >= (4 * $(HOUR)))
#RunBenchmarks : False

##  Normally, when the 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 startd might miss
##  keyboard activity by doing this.  So, if your utmp is unreliable,
##  set this setting to True and the startd will check the access time
##  on all tty and pty devices.
#STARTD_HAS_BAD_UTMP = False

##  This entry allows the 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 startd's ClassAd.  Just give a comma-separated list of the
##  names of devices you want considered the console, without the
##  "/dev/" portion of the pathname.
CONSOLE_DEVICES = mouse, console


##  The STARTD_ATTRS (and legacy STARTD_EXPRS) entry allows you to
##  have the startd advertise arbitrary attributes from the config
##  file in its ClassAd.  Give the comma-separated list of entries
##  from the config file you want in the startd ClassAd.
##  NOTE: because of the different syntax of the config file and
##  ClassAds, you might have to do a little extra work to get a given
##  entry into the ClassAd.  In particular, ClassAds require double
##  quotes (") around your strings.  Numeric values can go in
##  directly, as can boolean expressions.  For example, if you wanted
##  the startd to advertise its list of console devices, when it's
##  configured to run benchmarks, and how often it sends updates to
##  the central manager, you'd have to define the following helper
##  macro:
#MY_CONSOLE_DEVICES = "$(CONSOLE_DEVICES)"
##  Note: this must come before you define STARTD_ATTRS because macros
##  must be defined before you use them in other macros or
##  expressions.
##  Then, you'd set the STARTD_ATTRS setting to this:
#STARTD_ATTRS = MY_CONSOLE_DEVICES, RunBenchmarks, UPDATE_INTERVAL
##
##  STARTD_ATTRS can also be defined on a per-slot basis.  The startd
##  builds the list of attributes to advertise by combining the lists
##  in this order: STARTD_ATTRS, SLOTx_STARTD_ATTRS.  In the below
##  example, the startd ad for slot1 will have the value for
##  favorite_color, favorite_season, and favorite_movie, and slot2
##  will have favorite_color, favorite_season, and favorite_song.
##
#STARTD_ATTRS = favorite_color, favorite_season
#SLOT1_STARTD_ATTRS = favorite_movie
#SLOT2_STARTD_ATTRS = favorite_song
##
##  Attributes in the STARTD_ATTRS list can also be on a per-slot basis.
##  For example, the following configuration:
##
#favorite_color = "blue"
#favorite_season = "spring"
#SLOT2_favorite_color = "green"
#SLOT3_favorite_season = "summer"
#STARTD_ATTRS = favorite_color, favorite_season
##
##  will result in the following attributes in the slot classified
##  ads:
##
## slot1 - favorite_color = "blue"; favorite_season = "spring"
## slot2 - favorite_color = "green"; favorite_season = "spring"
## slot3 - favorite_color = "blue"; favorite_season = "summer"
##
##  Finally, the recommended default value for this setting, is to
##  publish the COLLECTOR_HOST setting as a string.  This can be
##  useful using the "$$(COLLECTOR_HOST)" syntax in the submit file
##  for jobs to know (for example, via their environment) what pool
##  they're running in.
COLLECTOR_HOST_STRING = "$(COLLECTOR_HOST)"
STARTD_ATTRS = COLLECTOR_HOST_STRING

##  When the startd is claimed by a remote user, it can also advertise
##  arbitrary attributes from the ClassAd of the job its working on.
##  Just list the attribute names you want advertised.
##  Note: since this is already a ClassAd, you don't have to do
##  anything funny with strings, etc.  This feature can be turned off
##  by commenting out this setting (there is no default).
STARTD_JOB_EXPRS = ImageSize, ExecutableSize, JobUniverse, NiceUser

##  If you want to "lie" to Condor about how many CPUs your machine
##  has, you can use this setting to override Condor's automatic
##  computation.  If you modify this, you must restart the startd for
##  the change to take effect (a simple condor_reconfig will not do).
##  Please read the section on "condor_startd Configuration File
##  Macros" in the Condor Administrators Manual for a further
##  discussion of this setting.  Its use is not recommended.  This
##  must be an integer ("N" isn't a valid setting, that's just used to
##  represent the default).
#NUM_CPUS = N
```

```
##  If you never want Condor to detect more the "N" CPUs, uncomment this
##  line out. You must restart the startd for this setting to take
##  effect. If set to 0 or a negative number, it is ignored.
##  By default, it is ignored. Otherwise, it must be a positive
##  integer ("N" isn't a valid setting, that's just used to
##  represent the default).
#MAX_NUM_CPUS = N

##  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.  For example:
#MEMORY = 128

##  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.
#RESERVED_MEMORY = 0

######
##  SMP startd settings
##
##  By default, Condor will evenly divide the resources in an SMP
##  machine (such as RAM, swap space and disk space) among all the
##  CPUs, and advertise each CPU as its own slot with an even share of
##  the system resources.  If you want something other than this,
##  there are a few options available to you.  Please read the section
##  on "Configuring The Startd for SMP Machines" in the Condor
##  Administrator's Manual for full details.  The various settings are
##  only briefly listed and described here.
######

##  The maximum number of different slot types.
#MAX_SLOT_TYPES = 10

##  Use this setting to define your own slot types.  This
##  allows you to divide system resources unevenly among your CPUs.
##  You must use a different setting for each different type you
##  define.  The "<N>" in the name of the macro listed below must be
##  an integer from 1 to MAX_SLOT_TYPES (defined above),
##  and you use this number to refer to your type.  There are many
##  different formats these settings can take, so be sure to refer to
##  the section on "Configuring The Startd for SMP Machines" in the
##  Condor Administrator's Manual for full details.  In particular,
##  read the section titled "Defining Slot Types" to help
##  understand this setting.  If you modify any of these settings, you
##  must restart the condor_start for the change to take effect.
#SLOT_TYPE_<N> = 1/4
#SLOT_TYPE_<N> = cpus=1, ram=25%, swap=1/4, disk=1/4
#  For example:
#SLOT_TYPE_1 = 1/8
#SLOT_TYPE_2 = 1/4

##  If you define your own slot types, you must specify how
##  many slots of each type you wish to advertise.  You do
##  this with the setting below, replacing the "<N>" with the
##  corresponding integer you used to define the type above.  You can
##  change the number of a given type being advertised at run-time,
##  with a simple condor_reconfig.
#NUM_SLOTS_TYPE_<N> = M
#  For example:
```

```
#NUM_SLOTS_TYPE_1 = 6
#NUM_SLOTS_TYPE_2 = 1

##  The number of evenly-divided slots you want Condor to
##  report to your pool (if less than the total number of CPUs).  This
##  setting is only considered if the "type" settings described above
##  are not in use.  By default, all CPUs are reported.  This setting
##  must be an integer ("N" isn't a valid setting, that's just used to
##  represent the default).
#NUM_SLOTS = N

##  How many of the slots the 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).  This must be an integer ("N" isn't a valid
##  setting, that's just used to represent the default).
#SLOTS_CONNECTED_TO_CONSOLE = N

##  How many of the slots the startd is representing should
##  be "connected" to the keyboard (for remote tty activity, as well
##  as console activity).  Defaults to 1.
#SLOTS_CONNECTED_TO_KEYBOARD = 1

##  If there are slots that aren't connected to the
##  keyboard or the console (see the above two settings), the
##  corresponding idle time reported will be the time since the startd
##  was spawned, plus the value of this parameter.  It defaults to 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 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 don't want
##  this boost, just 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, just increase this parameter.
#DISCONNECTED_KEYBOARD_IDLE_BOOST = 1200

######
##  Settings for computing optional resource availability statistics:
######
##  If STARTD_COMPUTE_AVAIL_STATS = True, the startd will compute
##  statistics about resource availability to be included in the
##  classad(s) sent to the collector describing the resource(s) the
##  startd manages.  The following attributes will always be included
##  in the resource classad(s) if STARTD_COMPUTE_AVAIL_STATS = True:
##    AvailTime = What proportion of the time (between 0.0 and 1.0)
##      has this resource been in a state other than "Owner"?
##    LastAvailInterval = What was 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 = At what time did the resource last leave the
##      "Owner" state?  Measured in the number of seconds since the
##      epoch (00:00:00 UTC, Jan 1, 1970).
##    AvailTimeEstimate = Based on past history, this is an estimate
##      of how long the current period between "Owner" states will
##      last.
#STARTD_COMPUTE_AVAIL_STATS = False

##  If STARTD_COMPUTE_AVAIL_STATS = True, STARTD_AVAIL_CONFIDENCE sets
##  the confidence level of the AvailTimeEstimate.  By default, the
##  estimate is based on the 80th percentile of past values.
#STARTD_AVAIL_CONFIDENCE = 0.8
```

```
##  STARTD_MAX_AVAIL_PERIOD_SAMPLES limits the number of samples of
##  past available intervals stored by the startd to limit memory and
##  disk consumption.  Each sample requires 4 bytes of memory and
##  approximately 10 bytes of disk space.
#STARTD_MAX_AVAIL_PERIOD_SAMPLES = 100

## CKPT_PROBE is the location of a program which computes aspects of the
## CheckpointPlatform classad attribute. By default the location of this
## executable will be here: $(LIBEXEC)/condor_ckpt_probe
CKPT_PROBE = $(LIBEXEC)/condor_ckpt_probe

##--------------------------------------------------------------------
##  condor_schedd
##--------------------------------------------------------------------
##  Where are the various shadow binaries installed?
SHADOW_LIST = SHADOW, SHADOW_STANDARD
SHADOW   = $(SBIN)/condor_shadow
SHADOW_STANDARD  = $(SBIN)/condor_shadow.std

##  When the schedd starts up, it can place it's address (IP and port)
##  into a file.  This way, tools running on the local machine don't
##  need to query the central manager to find the schedd.  This
##  feature can be turned off by commenting out this setting.
SCHEDD_ADDRESS_FILE = $(SPOOL)/.schedd_address

##  Additionally, a daemon may store its ClassAd on the local filesystem
##  as well as sending it to the collector. This way, tools that need
##  information about a daemon do not have to contact the central manager
##  to get information about a daemon on the same machine.
##  This feature is necessary for Quill to work.
SCHEDD_DAEMON_AD_FILE = $(SPOOL)/.schedd_classad

##  How often should the schedd send an update to the central manager?
#SCHEDD_INTERVAL = 300

##  How long should the schedd wait between spawning each shadow?
#JOB_START_DELAY = 2

##  How many concurrent sub-processes should the schedd spawn to handle
##  queries?  (Unix only)
#SCHEDD_QUERY_WORKERS   = 3

##  How often should the schedd send a keep alive message to any
##  startds it has claimed?  (5 minutes)
#ALIVE_INTERVAL  = 300

##  This setting controls the maximum number of times that a
##  condor_shadow processes can have a fatal error (exception) before
##  the condor_schedd will simply relinquish the match associated with
##  the dying shadow.
#MAX_SHADOW_EXCEPTIONS = 5

##  Estimated virtual memory size of each condor_shadow process.
##  Specified in kilobytes.
# SHADOW_SIZE_ESTIMATE = 800

##  The condor_schedd can renice the condor_shadow processes on your
##  submit machines.  How "nice" do you want the shadows? (1-19).
##  The higher the number, the lower priority the shadows have.
# SHADOW_RENICE_INCREMENT = 0

## The condor_schedd can renice scheduler universe processes
## (e.g. DAGMan) on your submit machines.  How "nice" do you want the
## scheduler universe processes? (1-19).  The higher the number, the
## lower priority the processes have.
# SCHED_UNIV_RENICE_INCREMENT = 0

##  By default, when the 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 NEGOTIATE_ALL_JOBS_IN_CLUSTER to True
##  will force the 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.
#NEGOTIATE_ALL_JOBS_IN_CLUSTER = False

## This setting controls how often, in seconds, the schedd considers
## periodic job actions given by the user in the submit file.
## (Currently, these are periodic_hold, periodic_release, and periodic_remove.)
#PERIODIC_EXPR_INTERVAL = 60

######
## Queue management settings:
######
##  How often should the schedd truncate it's job queue transaction
##  log?  (Specified in seconds, once a day is the default.)
#QUEUE_CLEAN_INTERVAL = 86400

##  How often should the schedd commit "wall clock" run time for jobs
##  to the queue, so run time statistics remain accurate when the
##  schedd crashes?  (Specified in seconds, once per hour is the
##  default.  Set to 0 to disable.)
#WALL_CLOCK_CKPT_INTERVAL = 3600

##  What users do you want to grant super user access to this job
##  queue?  (These users will be able to remove other user's jobs).
##  By default, this only includes root.
QUEUE_SUPER_USERS = root, condor


##--------------------------------------------------------------------
##  condor_shadow
##--------------------------------------------------------------------
##  If the shadow is unable to read a checkpoint file from the
##  checkpoint server, it keeps trying only if the job has accumulated
##  more than MAX_DISCARDED_RUN_TIME seconds of CPU usage.  Otherwise,
##  the job is started from scratch.  Defaults to 1 hour.  This
##  setting is only used if USE_CKPT_SERVER (from above) is True.
#MAX_DISCARDED_RUN_TIME = 3600

##  Should periodic checkpoints be compressed?
#COMPRESS_PERIODIC_CKPT = False

##  Should vacate checkpoints be compressed?
#COMPRESS_VACATE_CKPT = False

##  Should we commit the application's dirty memory pages to swap
##  space during a periodic checkpoint?
#PERIODIC_MEMORY_SYNC = False
```

```
##  Should we write vacate checkpoints slowly?  If nonzero, this
##  parameter specifies the speed at which vacate checkpoints should
##  be written, in kilobytes per second.
#SLOW_CKPT_SPEED = 0

##  How often should the shadow update the job queue with job
##  attributes that periodically change?  Specified in seconds.
#SHADOW_QUEUE_UPDATE_INTERVAL = 15 * 60

##  Should the shadow wait to update certain job attributes for the
##  next periodic update, or should it immediately these update
##  attributes as they change?  Due to performance concerns of
##  aggressive updates to a busy condor_schedd, the default is True.
#SHADOW_LAZY_QUEUE_UPDATE = TRUE


##--------------------------------------------------------------------
##  condor_starter
##--------------------------------------------------------------------
##  The condor_starter can renice the processes from remote Condor
##  jobs on your execute machines.  If you want this, uncomment the
##  following entry and set it to how "nice" you want the user
##  jobs. (1-19)  The larger the number, the lower priority the
##  process gets on your machines.
##  Note on Win32 platforms, this number needs to be greater than
##  zero (i.e. the job must be reniced) or the mechanism that
##  monitors CPU load on Win32 systems will give erratic results.
#JOB_RENICE_INCREMENT = 10

##  Should the starter 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?
#STARTER_LOCAL_LOGGING = TRUE

##  If the UID_DOMAIN settings match on both the execute and submit
##  machines, but the UID of the user who submitted the job isn't in
##  the passwd file of the execute machine, the starter will normally
##  exit with an error.  Do you want the starter to just start up the
##  job with the specified UID, even if it's not in the passwd file?
#SOFT_UID_DOMAIN = FALSE


##--------------------------------------------------------------------
##  condor_procd
##--------------------------------------------------------------------
##
# the path to the procd binary
#
PROCD = $(SBIN)/condor_procd

# the path to the procd "address"
#   - on UNIX this will be a named pipe; we'll put it in the
#     $(LOCK) directory by default (note that multiple named pipes
#     will be created in this directory for when the procd responds
#     to its clients)
#   - on Windows, this will be a named pipe as well (but named pipes on
#     Windows are not even close to the same thing as named pipes on
#     UNIX); the name will be something like:
#         \\.\pipe\condor_procd
#
PROCD_ADDRESS = $(RUN)/procd_pipe

# The procd currently uses a very simplistic logging system. Since this
# log will not be rotated like other Condor logs, it is only recommended
# to set PROCD_LOG when attempting to debug a problem. In other Condor
# daemons, turning on D_PROCFAMILY will result in that daemon logging
# all of its interactions with the ProcD.
#
#PROCD_LOG = $(LOG)/ProcLog

# This is the maximum period that the procd will use for taking
# snapshots (the actual period may be lower if a condor daemon registers
# a family for which it wants more frequent snapshots)
#
PROCD_MAX_SNAPSHOT_INTERVAL = 60

# On Windows, we send a process a "soft kill" via a WM_CLOSE message.
# This binary is used by the ProcD (and other Condor daemons if PRIVSEP
# is not enabled) to help when sending soft kills.
WINDOWS_SOFTKILL = $(SBIN)/condor_softkill

##--------------------------------------------------------------------
##  condor_submit
##--------------------------------------------------------------------
##  If you want condor_submit to automatically append an expression to
##  the Requirements expression or Rank expression of jobs at your
##  site, uncomment these entries.
#APPEND_REQUIREMENTS = (expression to append job requirements)
#APPEND_RANK  = (expression to append job rank)

##  If you want expressions only appended for either standard or
##  vanilla universe jobs, you can uncomment these entries.  If any of
##  them are defined, they are used for the given universe, instead of
##  the generic entries above.
#APPEND_REQ_VANILLA = (expression to append to vanilla job requirements)
#APPEND_REQ_STANDARD = (expression to append to standard job requirements)
#APPEND_RANK_STANDARD = (expression to append to vanilla job rank)
#APPEND_RANK_VANILLA = (expression to append to standard job rank)

##  This can be used to define a default value for the rank expression
##  if one is not specified in the submit file.
#DEFAULT_RANK         = (default rank expression for all jobs)

##  If you want universe-specific defaults, you can use the following
##  entries:
#DEFAULT_RANK_VANILLA = (default rank expression for vanilla jobs)
#DEFAULT_RANK_STANDARD = (default rank expression for standard jobs)

##  If you want condor_submit to automatically append expressions to
##  the job ClassAds it creates, you can uncomment and define the
##  SUBMIT_EXPRS setting.  It works just like the STARTD_EXPRS
##  described above with respect to ClassAd vs. config file syntax,
##  strings, etc.  One common use would be to have the full hostname
##  of the machine where a job was submitted placed in the job
##  ClassAd.  You would do this by uncommenting the following lines:
#MACHINE = "$(FULL_HOSTNAME)"
#SUBMIT_EXPRS = MACHINE

## 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.
#DEFAULT_IO_BUFFER_SIZE = 524288

## Condor will attempt to consolidate small read and write operations
## into large blocks.  This macro specifies the default block size
## Condor will use.
#DEFAULT_IO_BUFFER_BLOCK_SIZE = 32768
```

```
##--------------------------------------------------------------------
##  condor_preen
##--------------------------------------------------------------------
##  Who should condor_preen send email to?
#PREEN_ADMIN  = $(CONDOR_ADMIN)

##  What files should condor_preen leave in the spool directory?
VALID_SPOOL_FILES = job_queue.log, job_queue.log.tmp, history, \
                          Accountant.log, Accountantnew.log, \
                          local_univ_execute, .quillwritepassword, \
        .pgpass, \
     .schedd_address, .schedd_classad

##  What files should condor_preen remove from the log directory?
INVALID_LOG_FILES = core

##--------------------------------------------------------------------
##  Java parameters:
##--------------------------------------------------------------------
##  If you would like this machine to be able to run Java jobs,
##  then set JAVA to the path of your JVM binary.  If you are not
##  interested in Java, there is no harm in leaving this entry
##  empty or incorrect.

JAVA = /usr/bin/java

##  Some JVMs need to be told the maximum amount of heap memory
##  to offer to the process.  If your JVM supports this, give
##  the argument here, and Condor will fill in the memory amount.
##  If left blank, your JVM will choose some default value,
##  typically 64 MB.  The default (-Xmx) works with the Sun JVM.

JAVA_MAXHEAP_ARGUMENT = -Xmx

## JAVA_CLASSPATH_DEFAULT gives the default set of paths in which
## Java classes are to be found.  Each path is separated by spaces.
## If your JVM needs to be informed of additional directories, add
## them here.  However, do not remove the existing entries, as Condor
## needs them.

JAVA_CLASSPATH_DEFAULT = $(SHARE) $(SHARE)/scimark2lib.jar .

##  JAVA_CLASSPATH_ARGUMENT describes the command-line parameter
##  used to introduce a new classpath:

JAVA_CLASSPATH_ARGUMENT = -classpath

##  JAVA_CLASSPATH_SEPARATOR describes the character used to mark
##  one path element from another:

JAVA_CLASSPATH_SEPARATOR = :

##  JAVA_BENCHMARK_TIME describes the number of seconds for which
##  to run Java benchmarks.  A longer time yields a more accurate
##  benchmark, but consumes more otherwise useful CPU time.
##  If this time is zero or undefined, no Java benchmarks will be run.

JAVA_BENCHMARK_TIME = 2

##  If your JVM requires any special arguments not mentioned in
##  the options above, then give them here.

JAVA_EXTRA_ARGUMENTS =
##
##--------------------------------------------------------------------
##  Condor-G settings
##--------------------------------------------------------------------
##  Where is the GridManager binary installed?

GRIDMANAGER   = $(SBIN)/condor_gridmanager
GT2_GAHP   = $(SBIN)/gahp_server
GRID_MONITOR   = $(SBIN)/grid_monitor.sh

##--------------------------------------------------------------------
##  Settings that control the daemon's debugging output:
##--------------------------------------------------------------------
##
## Note that the Gridmanager runs as the User, not a Condor daemon, so
## all users must have write permssion to the directory that the
## Gridmanager will use for it's logfile. Our suggestion is to create a
## directory called GridLogs in $(LOG) with UNIX permissions 1777
## (just like /tmp )
##  Another option is to use /tmp as the location of the GridManager log.
##

MAX_GRIDMANAGER_LOG = 1000000
GRIDMANAGER_DEBUG =

GRIDMANAGER_LOG = $(LOG)/GridmanagerLog.$(USERNAME)
GRIDMANAGER_LOCK = $(LOCK)/GridmanagerLock.$(USERNAME)

##--------------------------------------------------------------------
##  Various other settings that the Condor-G can use.
##--------------------------------------------------------------------

## For grid-type gt2 jobs (pre-WS GRAM), limit the number of jobmanager
## processes the gridmanager will let run on the headnode. Letting too
## many jobmanagers run causes severe load on the headnode.
GRIDMANAGER_MAX_JOBMANAGERS_PER_RESOURCE = 10

## If we're talking to a Globus 2.0 resource, Condor-G will use the new
## version of the GRAM protocol. The first option is how often to check the
## proxy on the submit site of things. If the GridManager discovers a new
## proxy, it will restart itself and use the new proxy for all future
## jobs launched. In seconds,  and defaults to 10 minutes
#GRIDMANAGER_CHECKPROXY_INTERVAL = 600

## The GridManager will shut things down 3 minutes before loosing Contact
## because of an expired proxy.
## In seconds, and defaults to 3 minutes
#GRDIMANAGER_MINIMUM_PROXY_TIME  = 180

## Condor requires that each submitted job be designated to run under a
## particular "universe".
##
## If no universe is specificed in the submit file, Condor must pick one
## for the job to use. By default, it chooses the "vanilla" universe.
## The default can be overridden in the config file with the DEFAULT_UNIVERSE
## setting, which is a string to insert into a job submit description if the
## job does not try and define it's own universe
##
#DEFAULT_UNIVERSE = vanilla

#
# The Cred_min_time_left is the first-pass at making sure that Condor-G
# does not submit your job without it having enough time left for the
# job to finish. For example, if you have a job that runs for 20 minutes, and
# you might spend 40 minutes in the queue, it's a bad idea to submit with less
# than an hour left before your proxy expires.
# 2 hours seemed like a reasonable default.
#
CRED_MIN_TIME_LEFT  = 120


##
## The GridMonitor allows you to submit many more jobs to a GT2 GRAM server
## than is normally possible.
#ENABLE_GRID_MONITOR = TRUE

##
## When an error occurs with the GridMonitor, how long should the
## gridmanager wait before trying to submit a new GridMonitor job?
## The default is 1 hour (3600 seconds).
#GRID_MONITOR_DISABLE_TIME = 3600

##
## The location of the wrapper for invoking
## Condor GAHP server
##
CONDOR_GAHP = $(SBIN)/condor_c-gahp
CONDOR_GAHP_WORKER = $(SBIN)/condor_c-gahp_worker_thread

##
## The Condor GAHP server has it's own log.  Like the Gridmanager, the
## GAHP server is run as the User, not a Condor daemon, so all users must
## have write permssion to the directory used for the logfile. Our
## suggestion is to create a directory called GridLogs in $(LOG) with
## UNIX permissions 1777 (just like /tmp )
## Another option is to use /tmp as the location of the CGAHP log.
##
MAX_C_GAHP_LOG = 1000000

#C_GAHP_LOG = $(LOG)/GridLogs/CGAHPLog.$(USERNAME)
C_GAHP_LOG = /tmp/CGAHPLog.$(USERNAME)
C_GAHP_LOCK = /tmp/CGAHPLock.$(USERNAME)
C_GAHP_WORKER_THREAD_LOG = /tmp/CGAHPWorkerLog.$(USERNAME)
C_GAHP_WORKER_THREAD_LOCK = /tmp/CGAHPWorkerLock.$(USERNAME)

##
## The location of the wrapper for invoking
## GT4 GAHP server
##
GT4_GAHP = $(SBIN)/gt4_gahp

##
## The location of GT4 files. This should normally be lib/gt4
##
GT4_LOCATION = $(LIB)/gt4

##
## The location of the wrapper for invoking
## GT4 GAHP server
##
GT42_GAHP = $(SBIN)/gt42_gahp

##
## The location of GT4 files. This should normally be lib/gt4
##
GT42_LOCATION = $(LIB)/gt42
```

```
##
## gt4 gram requires a gridftp server to perform file transfers.
## If GRIDFTP_URL_BASE is set, then Condor assumes there is a gridftp
## server set up at that URL suitable for its use. Otherwise, Condor
## will start its own gridftp servers as needed, using the binary
## pointed at by GRIDFTP_SERVER. GRIDFTP_SERVER_WRAPPER points to a
## wrapper script needed to properly set the path to the gridmap file.
##
#GRIDFTP_URL_BASE = gsiftp://$(FULL_HOSTNAME)
GRIDFTP_SERVER = $(LIBEXEC)/globus-gridftp-server
GRIDFTP_SERVER_WRAPPER = $(LIBEXEC)/gridftp_wrapper.sh

##
## Location of the PBS/LSF gahp and its associated binaries
##
GLITE_LOCATION = $(LIB)/glite
PBS_GAHP = $(GLITE_LOCATION)/bin/batch_gahp
LSF_GAHP = $(GLITE_LOCATION)/bin/batch_gahp

##
## The location of the wrapper for invoking the Unicore GAHP server
##
UNICORE_GAHP = $(SBIN)/unicore_gahp

##
## The location of the wrapper for invoking the NorduGrid GAHP server
##
NORDUGRID_GAHP = $(SBIN)/nordugrid_gahp

## The location of the CREAM GAHP server
CREAM_GAHP = $(SBIN)/cream_gahp

## Condor-G and CredD can use MyProxy to refresh GSI proxies which are
## about to expire.
#MYPROXY_GET_DELEGATION = /path/to/myproxy-get-delegation

##
## EC2: Universe = Grid, Grid_Resource = Amazon
##

## The location of the amazon_gahp program, required
AMAZON_GAHP = $(SBIN)/amazon_gahp

## Location of log files, useful for debugging, must be in
## a directory writable by any user, such as /tmp
#AMAZON_GAHP_DEBUG = D_FULLDEBUG
AMAZON_GAHP_LOG = /tmp/AmazonGahpLog.$(USERNAME)

## The number of seconds between status update requests to EC2. You can
## make this short (5 seconds) if you want Condor to respond quickly to
## instances as they terminate, or you can make it long (300 seconds = 5
## minutes) if you know your instances will run for awhile and don't mind
## delay between when they stop and when Condor responds to them
## stopping.
GRIDMANAGER_JOB_PROBE_INTERVAL = 300

## As of this writing Amazon EC2 has a hard limit of 20 concurrently
## running instances, so a limit of 20 is imposed so the GridManager
## does not waste its time sending requests that will be rejected.
GRIDMANAGER_MAX_SUBMITTED_JOBS_PER_RESOURCE_AMAZON = 20

##
##--------------------------------------------------------------------
##  condor_credd credential managment daemon
```

```
##--------------------------------------------------------------------
##  Where is the CredD binary installed?
CREDD    = $(SBIN)/condor_credd

##  When the credd starts up, it can place it's address (IP and port)
##  into a file.  This way, tools running on the local machine don't
##  need an additional "-n host:port" command line option.  This
##  feature can be turned off by commenting out this setting.
CREDD_ADDRESS_FILE = $(LOG)/.credd_address

##  Specify a remote credd server here,
#CREDD_HOST  = $(CONDOR_HOST):$(CREDD_PORT)

## CredD startup arguments
## Start the CredD on a well-known port.  Uncomment to to simplify
## connecting to a remote CredD.  Note: that this interface may change
## in a future release.
CREDD_PORT   = 9620
CREDD_ARGS   = -p $(CREDD_PORT) -f

## CredD daemon debugging log
CREDD_LOG   = $(LOG)/CredLog
CREDD_DEBUG   = D_FULLDEBUG
MAX_CREDD_LOG  = 4000000

## The credential owner submits the credential.  This list specifies
## other user who are also permitted to see all credentials.  Defaults
## to root on Unix systems, and Administrator on Windows systems.
#CRED_SUPER_USERS =

## Credential storage location.  This directory must exist
## prior to starting condor_credd.  It is highly recommended to
## restrict access permissions to _only_ the directory owner.
CRED_STORE_DIR = $(LOCAL_DIR)/cred_dir

## Index file path of saved credentials.
## This file will be automatically created if it does not exist.
#CRED_INDEX_FILE = $(CRED_STORE_DIR/cred-index

## condor_credd  will attempt to refresh credentials when their
## remaining lifespan is less than this value.  Units = seconds.
#DEFAULT_CRED_EXPIRE_THRESHOLD = 3600

## condor-credd periodically checks remaining lifespan of stored
## credentials, at this interval.
#CRED_CHECK_INTERVAL = 60

##
##--------------------------------------------------------------------
##  Stork data placment server
##--------------------------------------------------------------------
##  Where is the Stork binary installed?
STORK    = $(SBIN)/stork_server

##  When Stork starts up, it can place it's address (IP and port)
##  into a file.  This way, tools running on the local machine don't
##  need an additional "-n host:port" command line option.  This
##  feature can be turned off by commenting out this setting.
STORK_ADDRESS_FILE = $(LOG)/.stork_address

##  Specify a remote Stork server here,
#STORK_HOST  = $(CONDOR_HOST):$(STORK_PORT)

## STORK_LOG_BASE specifies the basename for heritage Stork log files.
## Stork uses this macro to create the following output log files:
## $(STORK_LOG_BASE): Stork server job queue classad collection
## journal file.
## $(STORK_LOG_BASE).history: Used to track completed jobs.
## $(STORK_LOG_BASE).user_log: User level log, also used by DAGMan.
STORK_LOG_BASE  = $(LOG)/Stork

## Modern Condor DaemonCore logging feature.
STORK_LOG = $(LOG)/StorkLog
STORK_DEBUG = D_FULLDEBUG
MAX_STORK_LOG = 4000000

## Stork startup arguments
## Start Stork on a well-known port.  Uncomment to to simplify
## connecting to a remote Stork.  Note: that this interface may change
## in a future release.
#STORK_PORT   = 34048
STORK_PORT   = 9621
STORK_ARGS = -p $(STORK_PORT) -f -Serverlog $(STORK_LOG_BASE)

## Stork environment.  Stork modules may require external programs and
## shared object libraries.  These are located using the PATH and
## LD_LIBRARY_PATH environments.  Further, some modules may require
## further specific environments.  By default, Stork inherits a full
## environment when invoked from condor_master or the shell.  If the
## default environment is not adequate for all Stork modules, specify
## a replacement environment here.  This environment will be set by
## condor_master before starting Stork, but does not apply if Stork is
## started directly from the command line.
#STORK_ENVIRONMENT = TMP=/tmp;CONDOR_CONFIG=/special/config;PATH=/lib

## Limits the number of concurrent data placements handled by Stork.
#STORK_MAX_NUM_JOBS = 5

## Limits the number of retries for a failed data placement.
#STORK_MAX_RETRY = 5

## Limits the run time for a data placement job, after which the
## placement is considered failed.
#STORK_MAXDELAY_INMINUTES = 10

## Temporary credential storage directory used by Stork.
#STORK_TMP_CRED_DIR = /tmp

## Directory containing Stork modules.
#STORK_MODULE_DIR = $(LIBEXEC)

##
##--------------------------------------------------------------------
##  Quill Job Queue Mirroring Server
##--------------------------------------------------------------------
##  Where is the Quill binary installed and what arguments should be passed?
QUILL = $(SBIN)/condor_quill
#QUILL_ARGS =

# Where is the log file for the quill daemon?
QUILL_LOG = $(LOG)/QuillLog

# The identification and location of the quill daemon for local clients.
QUILL_ADDRESS_FILE = $(LOG)/.quill_address

# If this is set to true, then the rest of the QUILL arguments must be defined
# for quill to function. If it is Fase or left undefined, then quill will not
# be consulted by either the scheduler or the tools, but in the case of a
# remote quill query where the local client has quill turned off, but the
# remote client has quill turned on, things will still function normally.
#QUILL_ENABLED = TRUE

#
# If Quill is enabled, by default it will only mirror the current job
# queue into the database. For historical jobs, and classads from other
# sources, the SQL Log must be enabled.
#QUILL_USE_SQL_LOG=FALSE

#
# The SQL Log can be enabled on a per-daemon basis. For example, to collect
# historical job information, but store no information about execute machines,
# uncomment these two lines
#QUILL_USE_SQL_LOG = FALSE
#SCHEDD.QUILL_USE_SQL_LOG = TRUE

# This will be the name of a quill daemon using this config file. This name
# should not conflict with any other quill name--or schedd name.
#QUILL_NAME = quill@postgresql-server.machine.com

# The Postgreql server requires usernames that can manipulate tables. This will
# be the username associated with this instance of the quill daemon mirroring
# a schedd's job queue. Each quill daemon must have a unique username
# associated with it otherwise multiple quill daemons will corrupt the data
# held under an indentical user name.
#QUILL_DB_NAME = name_of_db

# The required password for the DB user which quill will use to read
# information from the database about the queue.
#QUILL_DB_QUERY_PASSWORD = foobar

# What kind of database server is this?
# For now, only PGSQL is supported
#QUILL_DB_TYPE = PGSQL

# The machine and port of the postgres server.
# Although this says IP Addr, it can be a DNS name.
# It must match whatever format you used for the .pgpass file, however
#QUILL_DB_IP_ADDR = machine.domain.com:5432

# The login to use to attach to the database for updating information.
# There should be an entry in file $SPOOL/.pgpass that gives the password
# for this login id.
#QUILL_DB_USER = quillwriter

# Polling period, in seconds, for when quill reads transactions out of the
# schedd's job queue log file and puts them into the database.
#QUILL_POLLING_PERIOD = 10

# Allows or disallows a remote query to the quill daemon and database
# which is reading this log file. Defaults to true.
#QUILL_IS_REMOTELY_QUERYABLE = TRUE

# Add debugging flags to here if you need to debug quill for some reason.
#QUILL_DEBUG = D_FULLDEBUG

# Number of seconds the master should wait for the Quill daemon to respond
# before killing it. This number might need to be increased for very
# large  logfiles.
# The default is 3600 (one hour), but kicking it up to a few hours won't hurt
#QUILL_NOT_RESPONDING_TIMEOUT = 3600

# Should Quill hold open a database connection to the DBMSD?
# Each open connection consumes resources at the server, so large pools
# (100 or more machines) should set this variable to FALSE. Note the
# default is TRUE.
#QUILL_MAINTAIN_DB_CONN = TRUE

##
##--------------------------------------------------------------------
##  Database Management Daemon settings
##--------------------------------------------------------------------
##  Where is the DBMSd binary installed and what arguments should be passed?
DBMSD = $(SBIN)/condor_dbmsd
DBMSD_ARGS = -f

# Where is the log file for the quill daemon?
DBMSD_LOG = $(LOG)/DbmsdLog

# Interval between consecutive purging calls (in seconds)
#DATABASE_PURGE_INTERVAL = 86400

# Interval between consecutive database reindexing operations
# This is only used when dbtype = PGSQL
#DATABASE_REINDEX_INTERVAL = 86400

# Number of days before purging resource classad history
# This includes things like machine ads, daemon ads, submitters
#QUILL_RESOURCE_HISTORY_DURATION = 7

# Number of days before purging job run information
# This includes job events, file transfers, matchmaker matches, etc
# This does NOT include the final job ad. condor_history does not need
# any of this information to work.
#QUILL_RUN_HISTORY_DURATION = 7

# Number of days before purging job classad history
# This is the information needed to run condor_history
#QUILL_JOB_HISTORY_DURATION = 3650

# DB size threshold for warning the condor administrator. This is checked
# after every purge. The size is given in gigabytes.
#QUILL_DBSIZE_LIMIT = 20

# Number of seconds the master should wait for the DBMSD to respond before
# killing it. This number might need to be increased for very large databases
# The default is 3600 (one hour).
#DBMSD_NOT_RESPONDING_TIMEOUT = 3600

##
##--------------------------------------------------------------------
##  VM Universe Parameters
##--------------------------------------------------------------------
## Where is the Condor VM-GAHP installed? (Required)
VM_GAHP_SERVER = $(SBIN)/condor_vm-gahp

## If the VM-GAHP is to have its own log, define
## the location of log file.
##
## Optionally, if you do NOT define VM_GAHP_LOG, logs of VM-GAHP will
## be stored in the starter's log file.
## However, on Windows machine you must always define VM_GAHP_LOG.
#
VM_GAHP_LOG = $(LOG)/VMGahpLog
MAX_VM_GAHP_LOG = 1000000
#VM_GAHP_DEBUG = D_FULLDEBUG
```

```
## What kind of virtual machine program will be used for
## the VM universe?
## The two options are vmware and xen.  (Required)
#VM_TYPE = vmware

## How much memory can be used for the VM universe? (Required)
## This value is the maximum amount of memory that can be used by the
## virtual machine program.
#VM_MEMORY = 128

## Want to support networking for VM universe?
## Default value is FALSE
#VM_NETWORKING = FALSE

## What kind of networking types are supported?
##
## If you set VM_NETWORKING to TRUE, you must define this parameter.
## VM_NETWORKING_TYPE = nat
## VM_NETWORKING_TYPE = bridge
## VM_NETWORKING_TYPE = nat, bridge
##
## If multiple networking types are defined, you may define
## VM_NETWORKING_DEFAULT_TYPE for default networking type.
## Otherwise, nat is used for default networking type.
## VM_NETWORKING_DEFAULT_TYPE = nat
#VM_NETWORKING_DEFAULT_TYPE = nat
#VM_NETWORKING_TYPE = nat

## In default, the number of possible virtual machines is same as
## NUM_CPUS.
## Since too many virtual machines can cause the system to be too slow
## and lead to unexpected problems, limit the number of running
## virtual machines on this machine with
#VM_MAX_NUMBER = 2

## When a VM universe job is started, a status command is sent
## to the VM-GAHP to see if the job is finished.
## If the interval between checks is too short, it will consume
## too much of the CPU. If the VM-GAHP fails to get status 5 times in a row,
## an error will be reported to startd, and then startd will check
## the availability of VM universe.
## Default value is 60 seconds and minimum value is 30 seconds
#VM_STATUS_INTERVAL = 60

## How long will we wait for a request sent to the VM-GAHP to be completed?
## If a request is not completed within the timeout, an error will be reported
## to the startd, and then the startd will check
## the availability of vm universe.  Default value is 5 mins.
#VM_GAHP_REQ_TIMEOUT = 300

## When VMware or Xen causes an error, the startd will disable the
## VM universe.  However, because some errors are just transient,
## we will test one more
## whether vm universe is still unavailable after some time.
## In default, startd will recheck vm universe after 10 minutes.
## If the test also fails, vm universe will be disabled.
#VM_RECHECK_INTERVAL = 600

## Usually, when we suspend a VM, the memory being used by the VM
## will be saved into a file and then freed.
## However, when we use soft suspend, neither saving nor memory freeing
## will occur.
## For VMware, we send SIGSTOP to a process for VM in order to
## stop the VM temporarily and send SIGCONT to resume the VM.
## For Xen, we pause CPU. Pausing CPU doesn't save the memory of VM
## into a file. It only stops the execution of a VM temporarily.
#VM_SOFT_SUSPEND = TRUE

## If Condor runs as root and a job comes from a different UID domain,
## Condor generally uses "nobody", unless SLOTx_USER is defined.
## If "VM_UNIV_NOBODY_USER" is defined, a VM universe job will run
## as the user defined in "VM_UNIV_NOBODY_USER" instead of "nobody".
##
## Notice: In VMware VM universe, "nobody" can not create a VMware VM.
## So we need to define "VM_UNIV_NOBODY_USER" with a regular user.
## For VMware, the user defined in "VM_UNIV_NOBODY_USER" must have a
## home directory.  So SOFT_UID_DOMAIN doesn't work for VMware VM universe job.
## If neither "VM_UNIV_NOBODY_USER" nor "SLOTx_VMUSER"/"SLOTx_USER" is defined,
## VMware VM universe job will run as "condor" instead of "nobody".
## As a result, the preference of local users for a VMware VM universe job
## which comes from the different UID domain is
## "VM_UNIV_NOBODY_USER" -> "SLOTx_VMUSER" -> "SLOTx_USER" -> "condor".
#VM_UNIV_NOBODY_USER = login name of a user who has home directory

## If Condor runs as root and "ALWAYS_VM_UNIV_USE_NOBODY" is set to TRUE,
## all VM universe jobs will run as a user defined in "VM_UNIV_NOBODY_USER".
#ALWAYS_VM_UNIV_USE_NOBODY = FALSE

##--------------------------------------------------------------------
##  VM Universe Parameters Specific to VMware
##--------------------------------------------------------------------

## Where is perl program? (Required)
VMWARE_PERL = perl

## Where is the Condor script program to control VMware? (Required)
VMWARE_SCRIPT = $(SBIN)/condor_vm_vmware.pl

## Networking parameters for VMware
##
## What kind of VMware networking is used?
##
## If multiple networking types are defined, you may specify different
## parameters for each networking type.
##
## Examples
## (e.g.) VMWARE_NAT_NETWORKING_TYPE = nat
## (e.g.) VMWARE_BRIDGE_NETWORKING_TYPE = bridged
##
##  If there is no parameter for specific networking type, VMWARE_NETWORKING_TYPE is used.
##
#VMWARE_NAT_NETWORKING_TYPE = nat
#VMWARE_BRIDGE_NETWORKING_TYPE = bridged
VMWARE_NETWORKING_TYPE = nat

## The contents of this file will be inserted into the .vmx file of
## the VMware virtual machine before Condor starts it.
#VMWARE_LOCAL_SETTINGS_FILE = /path/to/file

##--------------------------------------------------------------------
##  VM Universe Parameters common to libvirt controlled vm's (xen & kvm)
##--------------------------------------------------------------------

##  Where is the Condor script program to control Xen & KVM? (Required)
VM_SCRIPT = $(SBIN)/condor_vm_xen.sh

## Networking parameters for Xen & KVM
##
```

```
## This is the path to the XML helper command; the libvirt_simple_script.awk
## script just reproduces what Condor already does for the kvm/xen VM
## universe
LIBVIRT_XML_SCRIPT = $(LIBEXEC)/libvirt_simple_script.awk

## This is the optional debugging output file for the xml helper
## script.  Scripts that need to output debugging messages should
## write them to the file specified by this argument, which will be
## passed as the second command line argument when the script is
## executed

#LIBVRT_XML_SCRIPT_ARGS = /dev/stderr

##--------------------------------------------------------------------
##  VM Universe Parameters Specific to Xen
##--------------------------------------------------------------------

##  Where is bootloader for Xen domainU? (Required)
##
##  The bootloader will be used in the case that a kernel image includes
##  a disk image
#XEN_BOOTLOADER = /usr/bin/pygrub

## The contents of this file will be added to the Xen virtual machine
## description that Condor writes.
#XEN_LOCAL_SETTINGS_FILE = /path/to/file

##
##--------------------------------------------------------------------
##  condor_lease_manager lease manager daemon
##--------------------------------------------------------------------
##  Where is the LeaseManager binary installed?
LeaseManager   = $(SBIN)/condor_lease_manager

# Turn on the lease manager
#DAEMON_LIST   = $(DAEMON_LIST), LeaseManager

# The identification and location of the lease manager for local clients.
LeaseManger_ADDRESS_FILE = $(LOG)/.lease_manager_address

## LeaseManager startup arguments
#LeaseManager_ARGS  = -local-name generic

## LeaseManager daemon debugging log
LeaseManager_LOG  = $(LOG)/LeaseManagerLog
LeaseManager_DEBUG  = D_FULLDEBUG
MAX_LeaseManager_LOG  = 1000000

# Basic parameters
LeaseManager.GETADS_INTERVAL = 60
LeaseManager.UPDATE_INTERVAL = 300
LeaseManager.PRUNE_INTERVAL = 60
LeaseManager.DEBUG_ADS  = False

LeaseManager.CLASSAD_LOG = $(SPOOL)/LeaseManagerState
#LeaseManager.QUERY_ADTYPE = Any
#LeaseManager.QUERY_CONSTRAINTS = target.MyType == "SomeType"
#LeaseManager.QUERY_CONSTRAINTS = target.TargetType == "SomeType"

##
##--------------------------------------------------------------------
##  KBDD - keyboard activity detection daemon
##--------------------------------------------------------------------
##  When the KBDD starts up, it can place it's address (IP and port)
##  into a file.  This way, tools running on the local machine don't
##  need an additional "-n host:port" command line option.  This
##  feature can be turned off by commenting out this setting.
KBDD_ADDRESS_FILE = $(LOG)/.kbdd_address

##
##--------------------------------------------------------------------
##  condor_ssh_to_job
##--------------------------------------------------------------------
# NOTE: condor_ssh_to_job is not supported under Windows.

# Tell the starter (execute side) whether to allow the job owner or
# queue super user on the schedd from which the job was submitted to
# use condor_ssh_to_job to access the job interactively (e.g. for
# debugging).  TARGET is the job; MY is the machine.
#ENABLE_SSH_TO_JOB = true

# Tell the schedd (submit side) whether to allow the job owner or
# queue super user to use condor_ssh_to_job to access the job
# interactively (e.g. for debugging).  MY is the job; TARGET is not
# defined.
#SCHEDD_ENABLE_SSH_TO_JOB = true

# Command condor_ssh_to_job should use to invoke the ssh client.
# %h --> remote host
# %i --> ssh key file
# %k --> known hosts file
# %u --> remote user
# %x --> proxy command
# %% --> %
#SSH_TO_JOB_SSH_CMD = ssh -oUser=%u -oIdentityFile=%i -oStrictHostKeyChecking=yes -
oUserKnownHostsFile=%k -oGlobalKnownHostsFile=%k -oProxyCommand=%x %h

# Additional ssh clients may be configured.  They all have the same
# default as ssh, except for scp, which omits the %h:
#SSH_TO_JOB_SCP_CMD = scp -oUser=%u -oIdentityFile=%i -oStrictHostKeyChecking=yes -
oUserKnownHostsFile=%k -oGlobalKnownHostsFile=%k -oProxyCommand=%x

# Path to sshd
#SSH_TO_JOB_SSHD = /usr/sbin/sshd

# Arguments the starter should use to invoke sshd in inetd mode.
# %f --> sshd config file
# %% --> %
#SSH_TO_JOB_SSHD_ARGS = "-i -e -f %f"

# sshd configuration template used by condor_ssh_to_job_sshd_setup.
SSH_TO_JOB_SSHD_CONFIG_TEMPLATE = $(ETC)/condor_ssh_to_job_sshd_config_template

# Path to ssh-keygen
#SSH_TO_JOB_SSH_KEYGEN = /usr/bin/ssh-keygen

# Arguments to ssh-keygen
# %f --> key file to generate
# %% --> %
#SSH_TO_JOB_SSH_KEYGEN_ARGS = "-N '' -C '' -q -f %f -t rsa"

######################################################################
##
##  Condor HDFS
##
##  This is the default local configuration file for configuring Condor
##  daemon responsible for running services related to hadoop
##  distributed storage system.You should copy this file to the
##  appropriate location and customize it for your needs.
##
##  Unless otherwise specified, settings that are commented out show
##  the defaults that are used if you don't define a value.  Settings
##  that are defined here MUST BE DEFINED since they have no default
##  value.
##
######################################################################

######################################################################
## FOLLOWING MUST BE CHANGED
######################################################################

## The location for hadoop installation directory. The default location
## is under 'libexec' directory. The directory pointed by HDFS_HOME
## should contain a lib folder that contains all the required Jars necessary
## to run HDFS name and data nodes.
#HDFS_HOME = $(RELEASE_DIR)/libexec/hdfs

## The host and port for hadoop's name node. If this machine is the
## name node (see HDFS_SERVICES) then the specified port will be used
## to run name node.
HDFS_NAMENODE = example.com:9000
HDFS_NAMENODE_WEB = example.com:8000

## You need to pick one machine as name node by setting this parameter
## to HDFS_NAMENODE. The remaining machines in a storage cluster will
## act as data nodes (HDFS_DATANODE).
HDFS_SERVICES = HDFS_DATANODE

## The two set of directories that are required by HDFS are for name
## node (HDFS_NAMENODE_DIR) and data node (HDFS_DATANODE_DIR). The
## directory for name node is only required for a machine running
## name node service and  is used to store critical meta data for
## files. The data node needs its directory to store file blocks and
## their replicas.
HDFS_NAMENODE_DIR = /tmp/hadoop_name
HDFS_DATANODE_DIR = /scratch/tmp/hadoop_data

## Unlike name node address settings (HDFS_NAMENODE), that needs to be
## well known across the storage cluster, data node can run on any
## arbitrary port of given host.
#HDFS_DATANODE_ADDRESS = 0.0.0.0:0

####################################################################
## OPTIONAL
#####################################################################

## Sets the log4j debug level. All the emitted debug output from HDFS
## will go in 'hdfs.log' under $(LOG) directory.
#HDFS_LOG4J=DEBUG

## The access to HDFS services both name node and data node can be
## restricted by specifying IP/host based filters. By default settings
## from ALLOW_READ/ALLOW_WRITE and DENY_READ/DENY_WRITE
## are used to specify allow and deny list. The below two parameters can
## be used to override these settings. Read the Condor manual for
## specification of these filters.
## WARN: HDFS doesn't make any distinction between read or write based connection.
#HDFS_ALLOW=*
#HDFS_DENY=*

#Fully qualified name for Name node and Datanode class.
#HDFS_NAMENODE_CLASS=org.apache.hadoop.hdfs.server.namenode.NameNode
```

```
#HDFS_DATANODE_CLASS=org.apache.hadoop.hdfs.server.datanode.DataNode

## In case an old name for hdfs configuration files is required.
#HDFS_SITE_FILE = hadoop-site.xml
```

Example A.1. The default global configuration file

Appendix B. Codes

This section describes the various codes used throughout MRG Grid.

B.1. Job universe codes

These codes are used in job ClassAds to determine which universe to use:

Code	Universe	Details
5	Vanilla universe	Single process, non-relinked jobs
7	Scheduler universe	Jobs run under the **schedd**
9	Grid universe	Jobs managed by the **condor_gridmanager**
10	Java universe	Jobs for the Java Virtual Machine
11	Parallel universe	General parallel jobs
12	Local universe	A job run under the **schedd** using a starter

Table B.1. Job Universe Codes

B.2. Job status codes

These codes are used in job ClassAds to describe the job status:

Code	Short Description	Long description
0	U	Unexpected
1	I	Idle
2	R	Running
3	X	Removed
4	C	Completed
5	H	Held
6	E	Submission Error

Table B.2. Job Status Codes

B.3. Job notification codes

These codes are used in job ClassAds to determine notification frequency:

Code	Frequency
0	Never
1	Always
2	Complete
3	Error

Table B.3. Job Notification Codes

B.4. Shadow exit status codes

These codes are used by **`condor_shadow`** when exiting:

Code	Command	Description
`4`	**`JOB_EXCEPTION`**	The job exited with an exception
`44`	**`DPRINTF_ERROR`**	There is a fatal error with **`dprintf()`**
`100`	**`JOB_EXITED`**	The job exited
`102`	**`JOB_KILLED`**	The job was killed
`103`	**`JOB_COREDUMPED`**	The job was killed and a core file produced
`105`	**`JOB_NO_MEM`**	There was not enough memory to start **`condor_shadow`**
`106`	**`JOB_SHADOW_USAGE`**	Incorrect arguments were provided to **`condor_shadow`**
`107`	**`JOB_SHOULD_REQUEUE`**	Requeue the job to be run again
`108`	**`JOB_NOT_STARTED`**	Cannot connect to **`condor_startd`**, or the request was refused
`109`	**`JOB_BAD_STATUS`**	The job status was something other than *`RUNNING`* when it was started
`110`	**`JOB_EXEC_FAILED`**	Execution failed for an unknown reason
`112`	**`JOB_SHOULD_HOLD`**	Put the job on hold
`113`	**`JOB_SHOULD_REMOVE`**	Remove the job

Table B.4. Shadow Exit Status Codes

B.5. Job hold reason codes

These codes are used to determine why a job has been held:

Code	Error	Description
`0`	**`Unspecified`**	This error code is being deprecated
`1`	**`UserRequest`**	The user put the job on hold with **`condor_hold`**
`3`	**`JobPolicy`**	The periodic hold expression evaluated to *`TRUE`*
`4`	**`CorruptedCredential`**	The credentials for the job were invalid

Code	Error	Description
`5`	**`JobPolicyUndefined`**	A job policy expression (such as PeriodicHold) evaluated to *`UNDEFINED`*
`6`	**`FailedToCreateProcess`**	The **`condor_starter`** could not start the executable
`7`	**`UnableToOpenOutput`**	The standard output file for the job could not be opened
`8`	**`UnableToOpenInput`**	The standard input file for the job could not be opened
`9`	**`UnableToOpenOutputStream`**	The standard output stream for the job could not be opened
`10`	**`UnableToOpenInputStream`**	The standard input stream for the job could not be opened
`11`	**`InvalidTransferAck`**	An internal protocol error was encountered when transferring files
`12`	**`DownloadFileError`**	The **`condor_starter`** could not download the input files
`13`	**`UploadFileError`**	The **`condor_starter`** could not upload the output files
`14`	**`IwdError`**	The initial working directory of the job cannot be accessed
`15`	**`SubmittedOnHold`**	The user requested the job be submitted on hold
`16`	**`SpoolingInput`**	Input files are being spooled

Table B.5. Job Hold Reason Codes

Appendix C. Feature Metadata

This appendix contains a list of the metadata associated with various features, for use with the remote configuration feature.

```
BaseJobExecuter
  conflicts: None
  included: None
  depends: None
BaseScheduler
  conflicts: None
  depends: Master, NodeAccess
  included: BaseJobExecuter
CentralManager
  conflicts: None
  depends: NodeAccess
  included: Collector, Negotiator
Collector
  conflicts: None
  depends: Master, NodeAccess
  included: None
CommonUIDDomain
  conflicts: None
  depends: None
  included: None
ConcurrencyLimits
  conflicts: None
  depends: None
  included: Negotiator
ConsoleCollector
  conflicts: None
  depends: QMF
  included: Collector
ConsoleScheduler
  conflicts: None
  depends: QMF, BaseScheduler
  included: None
ConsoleExecuteNode
  conflicts: None
  depends: QMF, ExecuteNode
  included: None
ConsoleMaster
  conflicts: None
  depends: QMF, Master
  included: None
ConsoleNegotiator
  conflicts: None
  depends: QMF, Negotiator
  included: None
DedicatedResource
  conflicts: None
  depends: None
  included: ExecuteNode
DedicatedScheduler
  conflicts: None
  depends: None
  included: Scheduler
DynamicSlots
  conflicts: None
  depends: None
  included: ExecuteNode
EC2
```

```
  conflicts: None
  depends: None
  included: ExecuteNode
EC2Enhanced
  conflicts: None
  depends: None
  included: JobRouter
ExecuteNode
  conflicts: None
  depends: Master
  included: BaseJobExecuter
ExecuteNodeDedicatedPreemption
  conflicts: None
  depends: None
  included: DedicatedResource
ExecuteNodeTriggerData
  conflicts: None
  depends: None
  included: ExecuteNode
HACentralManager
  conflicts: None
  depends: None
  included: CentralManager
HAScheduler
  conflicts: Scheduler
  depends: None
  included: JobQueueLocation, BaseScheduler
JobHooks
  conflicts: None
  depends: None
  included: None
JobQueueLocation
  conflicts: None
  depends: None
  included: None
JobRouter
  conflicts: None
  depends: Master, BaseScheduler
  included: None
JobServer
  conflicts: None
  depends: Master, QMF, JobQueueLocation
  included: None
KeyboardMonitor
  conflicts: None
  depends: Master
  included: None
LowLatency
  conflicts: None
  depends: JobHooks
  included: ExecuteNode
Master
  conflicts: None
  depends: NodeAccess
  included: None
Negotiator
  conflicts: None
  depends: Master, NodeAccess
  included: None
NodeAccess
  conflicts: None
  depends: None
  included: None
QMF
```

```
  conflicts: None
  depends: None
  included: None
Scheduler
  conflicts: HAScheduler
  depends: None
  included: JobQueueLocation, BaseScheduler
SchedulerDedicatedPreemption
  conflicts: None
  depends: None
  included: DedicatedScheduler
SharedFileSystem
  conflicts: None
  depends: None
  included: None
TriggerService
  conflicts: None
  depends: Master, QMF
  included: None
VMUniverse
  conflicts: None
  depends: None
  included: ExecuteNode
```

Note

The **`BaseJobExecuter`** and **`BaseScheduler`** features are not intended to be installed alone. They must be installed with a feature that depends on them.

Appendix D. Revision History

Revision 7.3 **Fri Oct 1 2010** **Lana Brindley** *lbrindle@redhat.com*

Minor XML correction

Revision 7.2 **Fri Oct 1 2010** **Lana Brindley** *lbrindle@redhat.com*

Minor XML correction

Revision 7.1 **Wed Sep 29 2010** **Lana Brindley** *lbrindle@redhat.com*

Minor error correction

Revision 7.0 **Tue Sep 28 2010** **Lana Brindley** *lbrindle@redhat.com*

Prepared for publishing

Revision 6.24 **Tue Sep 28 2010** **Lana Brindley** *lbrindle@redhat.com*

BZ#637773 & #637771 - DAGMan chapter

Revision 6.23 **Fri Sep 24 2010** **Lana Brindley** *lbrindle@redhat.com*

BZ#636851 - Users chapter

Revision 6.22 **Wed Sep 22 2010** **Lana Brindley** *lbrindle@redhat.com*

BZ#632238 - Security chapter

Revision 6.21 **Tue Sep 21 2010** **Lana Brindley** *lbrindle@redhat.com*

Integrated QE changes from Luigi Toscano

Revision 6.20 **Tue Sep 21 2010** **Lana Brindley** *lbrindle@redhat.com*

BZ#632238 - Security chapter
BZ#632024 - Configuration chapter
Minor updates from rrati on IRC

Revision 6.19 **Mon Sep 20 2010** **Lana Brindley** *lbrindle@redhat.com*

BZ#634742 - Remote Configuration chapter
BZ#632654 - Jobs chapter

Revision 6.18 **Wed Sep 15 2010** **Lana Brindley** *lbrindle@redhat.com*

Integrated QE changes from Lubos Trilety
Integrated QE changes from Luigi Toscano

BZ#632421 - Remote Configuration chapter
Prepared for MRG QE review

Revision 6.17 **Tue Sep 14 2010** **Lana Brindley** *lbrindle@redhat.com*

BZ#632024 - Configuration chapter
BZ#632238 - Security chapter
BZ#632355 - Remote Configuration chapter
BZ#632654 - Jobs chapter

Revision 6.16 **Mon Sep 13 2010** **Lana Brindley** *lbrindle@redhat.com*

BZ#618885 - Remote configuration chapter
BZ#631560 - Overview chapter
BZ#632024 - Configuration chapter

Revision 6.15 **Fri Sep 10 2010** **Lana Brindley** *lbrindle@redhat.com*

BZ#615002 - VM Universe chapter
BZ#618389 - Security chapter

Revision 6.14 **Wed Sep 8 2010** **Lana Brindley** *lbrindle@redhat.com*

BZ#628106 - Security chapter

Revision 6.13 **Wed Sep 8 2010** **Lana Brindley** *lbrindle@redhat.com*

BZ#628106 - Security chapter

Revision 6.12 **Wed Sep 8 2010** **Lana Brindley** *lbrindle@redhat.com*

BZ#621927 - Appendix A - Configuration
BZ#621977 - Appendix B - Codes
BZ#628106 - Security chapter

Revision 6.11 **Mon Sep 6 2010** **Lana Brindley** *lbrindle@redhat.com*

BZ#588502 - Appendix A - Configuration
BZ#615002 - VM Universe chapter
BZ#615036 - Concurrency Limits chapter
BZ#615043, BZ#628110 & BZ#626824 - Configuration chapter
BZ#615490 - DAGMan chapter
BZ#618885 & BZ#628105 - Remote Config chapter
BZ#628108 - Users chapter

Revision 6.10 **Thu Aug 26 2010** **Lana Brindley** *lbrindle@redhat.com*

BZ#625035 - Remote Config chapter
Integrated QE changes from Luigi Toscano

Integrated QE changes from Lubos Trilety
Prepared for technical review

Revision 6.9 **Wed Aug 25 2010** **Lana Brindley** *lbrindle@redhat.com*

BZ#596241 - Security chapter
BZ#607675 - FAQ chapter
BZ#615334 - Low Latency chapter
BZ#617252 - HFS info in Users chapter
BZ#626824 & #622797 - Configuration chapter

Revision 6.8 **Tue Aug 24 2010** **Lana Brindley** *lbrindle@redhat.com*

BZ#619588 & #618885 - Remote Configuration chapter
BZ#615002 - VM Universe chapter
BZ#561136 - Removed erroneous "dynamic provisioning" statement
BZ#621927 - Appendix A - Configuration
BZ#621977 - Appendix B - Codes

Revision 6.7 **Mon Aug 23 2010** **Lana Brindley** *lbrindle@redhat.com*

BZ#618389 - Security chapter

Revision 6.6 **Thu Aug 19 2010** **Lana Brindley** *lbrindle@redhat.com*

BZ#618389 - Security chapter

Revision 6.5 **Thu Aug 5 2010** **Lana Brindley** *lbrindle@redhat.com*

BZ#615043 - Windows chapter

Revision 6.4 **Fri Jul 30 2010** **Lana Brindley** *lbrindle@redhat.com*

BZ#615002 - VM Universe chapter

Revision 6.3 **Wed Jul 28 2010** **Lana Brindley** *lbrindle@redhat.com*

BZ#614498 - Remote Configuration chapter

Revision 6.2 **Tue Jul 27 2010** **Lana Brindley** *lbrindle@redhat.com*

Reviewed chapter layout

Revision 6.1 **Tue Jul 27 2010** **Lana Brindley** *lbrindle@redhat.com*

BZ#614498 - New Remote Configuration chapter

Revision 5.13 **Wed Jul 21 2010** **Lana Brindley** *lbrindle@redhat.com*

Minor change ahead of MRG QE review

Revision 5.12 **Tue Jul 20 2010** **Lana Brindley** *lbrindle@redhat.com*

Completed integrating results of technical review
Updated images
Prepared for MRG QE review

Revision 5.11 **Mon Jul 19 2010** **Lana Brindley** *lbrindle@redhat.com*

Integrated results of technical review

Revision 5.10 **Mon Jun 28 2010** **Lana Brindley** *lbrindle@redhat.com*

Updates for 1.3
BZ #602757 - updated hierarchical fair share info in Users chapter

Revision 5.9 **Thu Jun 24 2010** **Lana Brindley** *lbrindle@redhat.com*

Updates for 1.3
BZ #607348 - updated package info in Low Latency chapter

Revision 5.8 **Mon Jun 21 2010** **Lana Brindley** *lbrindle@redhat.com*

Updates for 1.3
BZ #603079 - removed info from EC2 chapter
BZ #603147 - corrected info in EC2 chapter
BZ #604172 - added info to Event Trigger chapter

Revision 5.7 **Fri Jun 18 2010** **Lana Brindley** *lbrindle@redhat.com*

Updates for 1.3
BZ #496560 - Added info to DAGman chapter

Revision 5.6 **Wed Jun 16 2010** **Lana Brindley** *lbrindle@redhat.com*

Updates for 1.3
BZ #580747 - Added custom kill signals info to Jobs and FAQ chapters
BZ #583093 - Added admonition to VM Universe chapter
BZ #584030 - Added information to Low Latency chapter
BZ #588502 & #593685 - Added config variables to Configuration Appendix

Revision 5.5 **Tue Jun 15 2010** **Lana Brindley** *lbrindle@redhat.com*

Updates for 1.3
BZ #510139 - Added Event Trigger chapter
BZ #533982 - Added Process Tracking chapter

Revision 5.4 **Wed Jun 9 2010** **Lana Brindley** *lbrindle@redhat.com*

Updates for 1.3
BZ #530580 - Configuration: MAX_FILE_DESCRIPTORS
BZ #533917 - MRG Grid FAQ contains references to capabilities that we do not ship
BZ #533974 - Grid user guide contains references to checkpointing, which we do not support
BZ #547181 - Partitionable Slot Defaults
BZ #547587 - Appendix A.2 - UNAME_ARCH contains superfluous text
BZ #561136 - Name "Dynamic Provisioning" needs to change
BZ #565619 - HA: append values to VALID_SPOOL_FILES, not overwrite it
BZ #577858 - EC2 Enhanced documentation updates
BZ #577867 - Updates to Low-Latency scheduling documentation
BZ #587423 - hook documentation does not state the privs each hook has
BZ #592406 - Sentence Fragment in Chapter 14

Revision 5.3 **Mon Dec 14 2009** **Lana Brindley** *lbrindle@redhat.com*

BZ #538085 - Changes to sample config

Revision 5.2 **Wed Dec 9 2009** **Lana Brindley** *lbrindle@redhat.com*

Replaced images

Revision 5.1 **Wed Dec 2 2009** **Lana Brindley** *lbrindle@redhat.com*

Removed broken images

Revision 5.0 **Thu Oct 29 2009** **Lana Brindley** *lbrindle@redhat.com*

Final version for 1.2 release

Revision 4.23 **Thu Oct 29 2009** **Lana Brindley** *lbrindle@redhat.com*

QE changes from ltoscano relating to KVM
Changed title of EC2 chapter
BZ #531375 - EC2 chapter edits
BZ #530995 - VM Universe chapter edits

Revision 4.22 **Thu Oct 22 2009** **Lana Brindley** *lbrindle@redhat.com*

QE changes from ltoscano relating to KVM

Revision 4.21 **Tue Oct 20 2009** **Lana Brindley** *lbrindle@redhat.com*

BZ #527490 - VM Universe chapter
QE changes from mkudlej
QE changes from ltoscano

Revision 4.20 **Fri Oct 16 2009** **Lana Brindley** *lbrindle@redhat.com*

BZ #529234 - EC2 chapter edits
BZ #529239 - Remote configuration chapter edits
BZ #529200 - Added formatTime description to ClassAds chapter

Revision 4.19 **Wed Oct 7 2009** **Lana Brindley** *lbrindle@redhat.com*

BZ #526123 - EC2 chapter edits

Revision 4.18 **Tue Oct 6 2009** **Lana Brindley** *lbrindle@redhat.com*

BZ #496562 - Remote configuration chapter edits
BZ #518655 - VM Universe chapter edits
BZ #520208 - EC2 chapter edits
BZ #525124 - HA chapter edits

Revision 4.17 **Thu Sep 24 2009** **Lana Brindley** *lbrindle@redhat.com*

BZ #482959 - Local configuration file location (added FAQ)
BZ #525093 - Updated caroniad and job hooks configuration file locations in EC2 chapter
BZ #525288 - Removed mention of caroniad and job hooks config files from low-latency chapter.
BZ #525281 - Removed Windows Execute Nodes chapter from view. This feature is not available in 1.2
BZ #525263 - Removed references to the backfill state from the Policy Configuration chapter
BZ #525090 - Changes to Low Latency chapter

Revision 4.16 **Mon Aug 24 2009** **Lana Brindley** *lbrindle@redhat.com*

Updated language in Dynamic slots and Windows-based execute nodes chapters

Revision 4.15 **Fri Aug 21 2009** **Lana Brindley** *lbrindle@redhat.com*

BZ #517580 & #518293 - Changes to Low-latency chapter
BZ #518260 & #517581 - Changes to EC2 chapter
BZ #496773 - New Windows execute nodes chapter
Changes in preparation for technical review

Revision 4.14 **Thu Aug 13 2009** **Lana Brindley** *lbrindle@redhat.com*

BZ #496569 & #513046 - Changes to EC2 Chapter
BZ #496571 - Added new FAQ
BZ #513505 - Changes to Low-latency chapter
BZ #470412 - Added X_CONSOLE_DISPLAY to Appendix A

Revision 4.13 **Tue Aug 11 2009** **Lana Brindley** *lbrindle@redhat.com*

BZ #496560 - Added DAGMan chapter

Revision 4.12 **Fri Jul 24 2009** **Lana Brindley** *lbrindle@redhat.com*

Reworking Low Latency chapter
Reworking APIs chapter
BZ #513219: Added new content to Low Latency chapter
BZ #496571: Added new FAQ
BZ #472362: Updated Appendix B with missing descriptions
BZ #485150: Updated Remote Configuration chapter with CNAME information
BZ #510162: Updated EC2 chapter with job hook information
BZ #510977: Added new Console chapter
BZ #513058: Reworked resource restriction info in ClassAds chapter

Revision 4.11 **Thu Jul 23 2009** **Lana Brindley** *`lbrindle@redhat.com`*

BZ #496563: Reworking Jobs chapter
BZ #496565: Reworking Users chapter
BZ #496568: Reworking High Availability chapter
Edited Concurrency Limits chapter
BZ #496570: Reworking Dynamic Configuration chapter

Revision 4.10 **Mon Jul 21 2009** **Lana Brindley** *`lbrindle@redhat.com`*

BZ #496561: moved example configuration files to end of Appendix A
BZ #496561: Reworking Configuration chapter
BZ #510138: Added condor_starter configuration variables to Appendix A

Revision 4.9 **Wed Jul 15 2009** **Lana Brindley** *`lbrindle@redhat.com`*

BZ #495659: Reworking EC2 chapter
BZ #496561: Reworking Configuration chapter
BZ #496561: Added example configuration files to Appendix A

Revision 4.8 **Fri Jul 10 2009** **Scott Mumford** *`smumford@redhat.com`*

Further additions to Appendix A

Revision 4.7 **Fri Jul 10 2009** **Lana Brindley** *`lbrindle@redhat.com`*

BZ #472362: Appendix B, codes
BZ #495659: Reworking EC2 chapter

Revision 4.6 **Tue Jul 7 2009** **Lana Brindley** *`lbrindle@redhat.com`*

BZ #471945: Low-latency example
Added link to Appendix A from Configuration.xml
Moved Policy_Configuration.xml to after Jobs.xml

Revision 4.5 **Fri Jun 26 2009** **Scott Mumford** *`smumford@redhat.com`*

Further additions to Appendix A

Revision 4.4 **Thu Jun 25 2009** **Scott Mumford** *smumford@redhat.com*
Further additions to Appendix A

Revision 4.3 **Wed Jun 24 2009** **Scott Mumford** *smumford@redhat.com*
Further additions to Appendix A

Revision 4.2 **Tue Jun 23 2009** **Scott Mumford** *smumford@redhat.com*
Added sections A.3 - A.7 to Appendix A

Revision 4.0 **Mon May 4 2009** **Lana Brindley** *lbrindle@redhat.com*
Copyedit Overview chapter
Moved "2.1. System wide configuration file variables" and "2.2. Logging configuration variables" into a new Appendix A

Revision 3.4 **Fri Mar 6 2009** **Lana Brindley** *lbrindle@redhat.com*
BZ#488852 - Added admonition to condor_submit -dump instructions in low latency chapter

Revision 3.3 **Thu Feb 26 2009** **Lana Brindley** *lbrindle@redhat.com*
BZ#484072 - Minor fixes to syntax in condor_configure_node

Revision 3.2 **Thu Feb 26 2009** **Lana Brindley** *lbrindle@redhat.com*
BZ#484072 - Update examples for condor_configure_node

Revision 3.1 **Fri Feb 13 2009** **Lana Brindley** *lbrindle@redhat.com*
BZ#484072 - New options for condor_configure_node
BZ#484045 - Update EC2 examples

Revision 3.0 **Tue Feb 10 2009** **Lana Brindley** *lbrindle@redhat.com*
Added information on EC2 Execute Node

Revision 22 **Mon Jan 19 2009** **Lana Brindley** *lbrindle@redhat.com*
Added links to product page

Revision 21 **Mon Jan 12 2009** **Lana Brindley** *lbrindle@redhat.com*
BZ #479198
BZ #473111

Revision 20 **Wed Jan 7 2009** **Lana Brindley** ***lbrindle@redhat.com***

BZ #479053

Revision 19 **Wed Jan 7 2009** **Lana Brindley** ***lbrindle@redhat.com***

BZ #477801
BZ #477805

Revision 18 **Mon Dec 22 2008** **Michael Hideo** ***mhideo@redhat.com***

BZ #477070
Removed issuenum in Book_Info.xml
Changed edition to 1

Revision 0.15 **Mon Dec 8 2008** **Lana Brindley** ***lbrindle***

BZ #474939
BZ #474938

Revision 0.14 **Fri Dec 5 2008** **Lana Brindley** ***lbrindle***

Further minor updates

Revision 0.13 **Tue Nov 25 2008** **Lana Brindley** ***lbrindle***

Further minor updates
Restructure of EC2 Chapter

Revision 0.12 **Mon Nov 24 2008** **Lana Brindley** ***lbrindle***

Minor updates prior to releasing document to Quality Engineering

Revision 0.11 **Mon Nov 24 2008** **Lana Brindley** ***lbrindle@redhat.com***

Completion of EC2 chapter

Revision 0.10 **Fri Nov 21 2008** **Lana Brindley** ***lbrindle@redhat.com***

Split EC2 chapter into EC2 and EC2 Enahnced - BZ #471695

Revision 0.9 **Thu Nov 20 2008** **Lana Brindley** ***lbrindle@redhat.com***

Added remote configuration chapter - BZ #471707

Revision 0.8 **Wed Nov 19 2008** **Lana Brindley** ***lbrindle@redhat.com***

Changes and updates arising from technical review

Revision 0.7	**Fri Nov 7 2008**	**Lana Brindley** *`lbrindle@redhat.com`*
Configuration		
Revision 0.6	**Mon Nov 3 2008**	**Lana Brindley** *`lbrindle@redhat.com`*
Concurrency limits - BZ #459937 Dynamic provisioning - BZ #468942 Low-latency scheduling - BZ #454455 FAQs More Information		
Revision 0.5	**Wed Oct 29 2008**	**Lana Brindley** *`lbrindle@redhat.com`*
Added download and configuration information to EC2 chapter APIs		
Revision 0.4	**Tue Oct 28 2008**	**Lana Brindley** *`lbrindle@redhat.com`*
EC2 Removed future chapters from current build		
Revision 0.3	**Tue Oct 21 2008**	**Lana Brindley** *`lbrindle@redhat.com`*
Policy Configuration Virtual Machine Universe High Availability		
Revision 0.2	**Wed Oct 1 2008**	**Lana Brindley** *`lbrindle@redhat.com`*
Front matter Preface Overview Configuration (not completed) Jobs Users ClassAds Policy Configuration (not completed)		
Revision 0.1	**Wed Aug 6 2008**	**Lana Brindley** *`lbrindle@redhat.com`*
Initial Document Creation		