15 Condor—A Distributed Job Scheduler

Todd Tannenbaum, Derek Wright, Karen Miller, and Miron Livny

Condor is a sophisticated and unique distributed job scheduler developed by the Condor research project at the University of Wisconsin-Madison Department of Computer Sciences.

A public domain version of the Condor software and complete documentation is freely available from the Condor project's website at `http://www.cs.wisc.edu/condor`. Organizations may optionally purchase a commercial version of Condor with an accompanying support contract; for additional information see URL `http://www.condorcomputing.com`.

This chapter introduces all aspects of Condor, from its ability to satisfy the needs and desires of both submitters and resource owners, to the management of Condor on clusters. Following an overview of Condor and Condor's ClassAd mechanism is a description of Condor from the user's perspective. The architecture of the software is presented along with overviews of installation and management. The chapter ends with configuration scenarios specific to clusters.

15.1 Introduction to Condor

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 jobs to Condor, and Condor places them into a queue, chooses when and where to run them based upon a policy, monitors their progress, and ultimately informs the user upon completion.

While providing functionality similar to that of a more traditional batch queueing system, Condor's novel architecture allows it to succeed in areas where traditional scheduling systems fail. Condor can be used to manage a cluster of dedicated Beowulf nodes. In addition, several unique mechanisms enable Condor to effectively harness wasted CPU power from otherwise idle desktop workstations. Condor can be used to seemlessly combine all of your organization's computational power into one resource.

Condor is the product of the Condor Research Project at the University of Wisconsin-Madison (UW-Madison), and was first installed as a production system in the UW-Madison Department of Computer Sciences nearly 10 years ago. This Condor installation has since served as a major source of computing cycles to UW-Madison faculty and students. Today, just in our department alone, Condor manages more than 1000 workstations including the department's 500 CPU Linux

Beowulf cluster. On a typical day, Condor delivers more than 650 CPU *days* to UW researchers. Additional Condor installations have been established over the years across our campus and the world. Hundreds of organizations in industry, government, and academia have used Condor to establish compute environments ranging in size from a handful to hundreds of workstations. We hope that Condor will help revolutionize your compute environment as well.

15.1.1 Features of Condor

Condor's features are extensive. It provides great flexibility for both the user submitting jobs and for the owner of a machine which provides CPU time towards running jobs. This list summarizes some of Condor's capabilities.

Distributed Submission. There is no single, centralized submission machine. Instead, Condor allows jobs to be submitted from many machines, and each machine contains its own job queue. Users may submit to a cluster from their own desktop machine.

Job Priorities. Users can assign priorities to their submitted jobs in order to control the execution order of the jobs. A "nice-user" mechanism requests the use of only those machines which would have otherwise been idle.

User Priorities. Administrators may assign priorities to users using a flexible mechanism which enables a policy of fair share, strict ordering, fractional ordering, or a combination of policies.

Job Dependence. Some sets of jobs require an ordering due to dependencies between jobs. Only start job X after jobs Y and Z successfully complete is an example of a dependency. Enforcing dependencies is easily handled.

Support for Multiple Job Models. Condor handles both serial jobs and parallel jobs incorporating PVM, dynamic PVM, and MPI.

ClassAds. The ClassAd mechanism in Condor provides an extremely flexible and expressive framework for matching resource requests (jobs) with resource offers (machines). Jobs can easily state both job requirements and job preferences. Likewise, machines can specify requirements and preferences about the jobs they are willing to run. These requirements and preferences can be described in powerful expressions, resulting in Condor's adaptation to nearly any desired policy.

Job Checkpoint and Migration. With certain types of jobs, Condor can transparently take a checkpoint and subsequently resume the application. A checkpoint is a snapshot of a job's complete state. Given a checkpoint, the job can later continue its execution from where it left off at the time of the checkpoint. A checkpoint also enables the transparent migration of a job from one machine to another machine.

Periodic Checkpoint. Condor can be configured to periodically produce a checkpoint for a job. This provides a form of fault tolerance and safeguards the accumulated computation time of a job. It reduces the loss in the event of a system failure such as the machine being shut down or hardware failure.

Job Suspend and Resume. Based upon policy rules, Condor can ask the operating system to suspend and later resume a job.

Remote System Calls. Despite running jobs on remote machines, Condor can often preserve the local execution environment via remote system calls. Users do not need to make data files available or even obtain a login account on remote workstations before Condor executes their programs there. The program behaves under Condor as if it were running as the user that submitted the job on the workstation where it was originally submitted, regardless of where it really executes.

Pools of Machines can Work Together. *Flocking* allows jobs to be scheduled across multiple Condor pools. It can be done across pools of machines owned by different organizations that impose their own policies.

Authentication and Authorization. Administrators have fine-grained control of access permissions, and Condor can perform strong network authentication using a variety of mechanisms including Kerberos and X.509 public key certificates.

Heterogeneous Platforms. In addition to Linux, Condor has been ported to most of the other primary flavors of Unix as well as Windows NT. A single pool can contain multiple platforms. Jobs to be executed under one platform may be submitted from a different platform. As an example, an executable that runs under Windows 2000 may be submitted from a machine running Linux.

Grid Computing. Condor incorporates many of the emerging Grid-based computing methodologies and protocols. It can interact with resources managed by Globus.

15.1.2 Understanding Condor ClassAds

The **ClassAd** is a flexible representation of the characteristics and constraints of both machines and jobs in the Condor system. **Matchmaking** is the mechanism by which Condor matches an idle job with an available machine. Understanding this unique framework is the key to harness the full flexibility of the Condor system.

ClassAds are employed by users to specify which machines should service their jobs. Administrators use them to customize scheduling policy.

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

All machines in a Condor pool advertise their attributes, such as available RAM memory, CPU type and speed, virtual memory size, current load average, current time and date, along with other static and dynamic properties. This machine ClassAd also advertises under what conditions it is willing to run a Condor job and what type of job it would prefer. These policy attributes can reflect the individual terms and preferences by which the different owners have graciously allowed their machines to participate in the Condor pool.

Upon submitting a job to Condor, a job ClassAd is created. This ClassAd includes attributes about the job, such as the amount of memory the job uses, the name of the program to run, the user who submitted the job, the time it was submitted, and much more. The job can also specify requirements and preferences (or **rank**) for the machine that will run the job. For instance, perhaps you are looking for the fastest floating point performance available. You want Condor to rank available machines based upon floating point performance. Perhaps you care only that the machine has a minimum of 256 Mbytes of RAM. Or, perhaps you will take any machine you can get! These job attributes and requirements are bundled up into a job ClassAd.

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

Structure of a ClassAd A ClassAd is a set of uniquely named expressions. Each named expression is called an *attribute*. Each attribute has an *attribute name* and an *attribute value*. The attribute value can be a simple integer, string, or floating point value, such as

```
Memory = 512
OpSys = "LINUX"
NetworkLatency = 7.5
```

An attribute value can also consist of a logical expression which will evaluate to TRUE, FALSE, or UNDEFINED. The syntax and operators allowed in these expressions are similar to those in C or Java, that is `==` for equals, `!=` for not-equals, `&&` for logical and, `||` for logical or, and so on. Furthermore, ClassAd expressions can incorporate attribute names to refer to other attribute values. For instance, consider the following small sample ClassAd:

```
MemoryInMegs = 512
MemoryInBytes = MemoryInMegs * 1024 * 1024
Cpus = 4
BigMachine = (MemoryInMegs > 256) && (Cpus >= 4)
VeryBigMachine = (MemoryInMegs > 512) && (Cpus >= 8)
FastMachine = BigMachine && SpeedRating
```

In this example, `BigMachine` evaluates to TRUE and `VeryBigMachine` evaluates to FALSE. But, because attribute `SpeedRating` is not specified, `FastMachine` would evaluate to UNDEFINED.

Condor provides *meta-operators* that allow you to explicitly compare against the UNDEFINED value by testing both the type and value of the operands. If both the types and values match, the two operands are considered *identical*. `=?=` is used for meta-equals, or is-identical-to, and `=!=` is used for meta-not-equals, or is-not-identical-to. These operators always return TRUE or FALSE, and therefore enable Condor administrators to specify explicit policies given incomplete information.

A complete description of ClassAd semantics and syntax is documented in the Condor Manual.

Matching ClassAds ClassAds can be matched with one another. This is the fundamental mechanism by which Condor matches jobs with machines. Figure 15.1 displays a ClassAd from Condor representing a machine and another representing a queued job. Each ClassAd contains a `MyType` attribute, describing what type of resource the ad represents, and a `TargetType` attribute. The `TargetType` specifies the type of resource desired in a match. Job ads want to be matched with machine ads and vice-versa.

Each ClassAd engaged in matchmaking specifies a `Requirements` and a `Rank` attribute. In order for two ClassAds to match, the `Requirements` expression in both ads must evaluate to TRUE. An important component of matchmaking is the

Job ClassAd	Machine ClassAd
MyType = “Job”	**MyType** = “Machine”
TargetType = “Machine”	**TargetType** = “Job”
Requirements = ((Arch==“INTEL” && OpSys==“LINUX”) && Disk > DiskUsage)	**Requirements** = Start
Rank = (Memory * 10000) + KFlops	**Rank** = TARGET.Department==MY.Department
Args = “-ini ./ies.ini”	**Activity** = “Idle”
ClusterId = 680	**Arch** = “INTEL”
Cmd = “/home/tannenba/bin/sim-exe”	**ClockDay** = 0
Department = “CompSci”	**ClockMin** = 614
DiskUsage = 465	**CondorLoadAvg** = 0.000000
StdErr = “sim.err”	**Cpus** = 1
ExitStatus = 0	**CurrentRank** = 0.000000
FileReadBytes = 0.000000	**Department** = “CompSci”
FileWriteBytes = 0.000000	**Disk** = 3076076
ImageSize = 465	**EnteredCurrentActivity** = 990371564
StdIn = “/dev/null”	**EnteredCurrentState** = 990330615
Iwd = “/home/tannenba/sim-m/run_55”	**FileSystemDomain** = “cs.wisc.edu”
JobPrio = 0	**IsInstructional** = FALSE
JobStartDate = 971403010	**KeyboardIdle** = 15
JobStatus = 2	**KFlops** = 145811
StdOut = “sim.out”	**LoadAvg** = 0.220000
Owner = “tannenba”	**Machine** = “nostos.cs.wisc.edu”
ProcId = 64	**Memory** = 511
QDate = 971377131	**Mips** = 732
RemoteSysCpu = 0.000000	**OpSys** = “LINUX”
RemoteUserCpu = 0.000000	**Start** = (LoadAvg <= 0.300000) && (KeyboardIdle > (15 * 60))
RemoteWallClockTime = 2401399.000000	**State** = “Unclaimed”
TransferFiles = “NEVER”	**Subnet** = “128.105.165”
WantCheckpoint = FALSE	**TotalVirtualMemory** = 787144
WantRemoteSyscalls = FALSE	⋮
⋮	

Figure 15.1
Examples of ClassAds in Condor.

`Requirements` and `Rank` expression can refer not only to attributes in their own ad, *but also to attributes in the candidate matching ad.* For instance, the `Requirements` expression for the job ad specified in Figure 15.1 refers to `Arch`, `OpSys`, and `Disk` which are all attributes found in the machine ad.

What if Condor finds more than one machine ClassAd which satisfies the `Requirements` constraint? That is where the `Rank` expression comes into play. The `Rank` expression specifies the desirability of the match (where higher numbers mean better matches). For example, the job ad in Figure 15.1 specifies:

```
Requirements = ((Arch=="INTEL" && OpSys=="LINUX") && Disk > DiskUsage)
Rank         = (Memory * 100000) + KFlops
```

In this case, the job requires a computer running the LINUX operating system and more local disk space than it will use. Among all such computers, the user prefers those with large physical memories and fast floating-point CPUs (`KFlops` is a metric of floating-point performance). Since the `Rank` is a user specified metric, *any* expression may be used to specify the perceived desirability of the match. Condor's matchmaking algorithms deliver the best resource (as defined by the `Rank` expression) while satisfying other criteria.

15.2 Using Condor

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

15.2.1 Roadmap to Using Condor

Here are the steps involved to run jobs using Condor:

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

Select the Condor Universe. Condor has five runtime environments from which to choose. Each runtime environment is called a *Universe*. Usually the Universe you choose is determined by the type of application you are asking Condor to run. There are six job Universes in total: two for serial jobs (Standard and Vanilla), one for parallel PVM jobs (PVM), one for parallel MPI jobs (MPI), one for Grid applications (GLOBUS), and one for meta-schedulers (Scheduler). Section 15.2.4 provides more information on each of these Universes.

Create a Submit Description file. The details of a job submission are defined in a *submit description* file. This file contains information about the job such as what executable to run, which Universe to use, the files to use for stdin, stdout, and stderr, requirements and preferences about the machine which should run the program, and where to send e-mail when the job completes. You can also tell Condor how many times to run a program; it is simple to run the same program multiple times with different data sets.

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

Once submitted, Condor handles all aspects of running the job. You can subsequently monitor the job's progress with the condor_q and condor_status commands. You may modify the order in which Condor will run your jobs with condor_prio. If desired, Condor can also record what is being done with your job at every stage in its lifecycle through the use of a log file specified during submission.

When the program completes, Condor notifies the owner (either by e-mail, the user-specified log file, or both) the exit status, along with various statistics, including time used and I/O performed. You can remove a job from the queue at any time with condor_rm.

15.2.2 Submitting a Job

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

The contents of a submit description file can save time for Condor users. It is easy to submit multiple runs of a program to Condor. To run the same program 500 times on 500 different input data sets, the data files are arranged such that each run reads its own input, and each run writes its own output. Every individual run may have its own initial working directory, stdin, stdout, stderr, command-line arguments, and shell environment.

Example submit description files illustrate the flexibility of using Condor. Assume the jobs submitted are serial jobs intended for a cluster that has a shared filesystem across all nodes. Therefore, all jobs use the Vanilla Universe, the simplest one for running serial jobs. The other Condor Universes are explored later.

Example 1: Very simple submit description file Example 1 is the simplest submit description file possible. It queues up one copy of the program 'foo' for execution by Condor. A log file called 'foo.log' is generated by Condor. The log file contains events pertaining to the job while it runs inside of Condor. When the job finishes, its exit conditions are noted in the log file. It is recommended that you always have a log file so you know what happened to your jobs. The *queue* statement in the submit description file tells Condor to use all the information specified so far to create a job ClassAd and place the job into the queue. Lines that begin with a pound character ('#') are comments and are ignored by `condor_submit`.

```
# Example 1 : Simple submit file
universe = vanilla
executable = foo
log = foo.log
queue
```

Example 2: More sophisticated submit description file Example 2 queues two copies of the program '`mathematica`'. The first copy runs in directory '`run_1`', and the second runs in directory '`run_2`'. For both queued copies, '`stdin`' will be '`test.data`', '`stdout`' will be '`loop.out`', and '`stderr`' will be '`loop.error`'. There will be two sets of files written, as the files are each written to their own directories. This is a convenient way to organize data for a large group of Condor jobs.

```
# Example 2: demonstrate use of multiple
# directories for data organization.
universe = vanilla
executable = mathematica
# Give some command line args, remap stdio
arguments = -solver matrix
input = test.data
output = loop.out
error = loop.error
log = loop.log

initialdir = run_1
queue
initialdir = run_2
queue
```

Example 3: Submit description file specifying 150 runs The submit description file for Example 3 queues 150 runs of program '`foo`'. This job requires Condor to run the program on machines which have greater than 128 megabytes of physical memory, and it further requires that the job not be scheduled to run on a specific node. Of the machines which meet the requirements, the job prefers to run on the fastest floating-point nodes currently available to accept the job. It also advises Condor that the job will use up to 180 megabytes of memory when running. Each of the 150 runs of the program is given its own process number, starting with process number 0. Several built-in macros can be used in a submit description file; one of them is the *$(Process)* macro which Condor expands to be the process number in the job cluster. This causes files '`stdin`', '`stdout`', and '`stderr`' to be '`in.0`', '`out.0`', and '`err.0`' for the first run of the program, '`in.1`', '`out.1`', and '`err.1`' for the second run of the program, and so forth. A single log file will list events for all 150 jobs in this job cluster.

```
# Example 3: Submit lots of runs and use the
# pre-defined $(Process) macro.
universe = vanilla
executable = foo
requirements = Memory > 128  && Machine != "server-node.cluster.edu"
rank = KFlops
image_size = 180

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

queue 150
```

Note that the `requirements` and `rank` entries in the submit description file will become the requirements and rank attributes of the subsequently created ClassAd for this job. These are arbitrary expressions that can reference any attributes of either the machine or the job; see Section 15.1.2 for more on requirements and rank expressions in ClassAds.

15.2.3 Overview of User Commands

Once you have jobs submitted to Condor, you can manage them and monitor their progress. Figure 15.1 shows several commands available to the Condor user to view the job queue, check the status of nodes in the pool, and several other activities. Most of these commands have many command-line options; please see the Command Reference chapter of the Condor Manual for complete documentation. To provide an introduction from a user perspective, we give a quick tour showing several of these commands in action.

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

```
% condor_status -submitters

Name                 Machine      Running IdleJobs HeldJobs

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

                           RunningJobs          IdleJobs           HeldJobs
```

Command	Description
condor_checkpoint	Checkpoint jobs running on the specified hosts
condor_compile	Create a relinked executable for submission to the Standard Universe
condor_glidein	Add a Globus resource to a Condor pool
condor_history	View log of Condor jobs completed to date
condor_hold	Put jobs in the queue in hold state
condor_prio	Change priority of jobs in the queue
condor_qedit	Modify attributes of a previously submitted job
condor_q	Display information about jobs in the queue
condor_release	Release held jobs in the queue
condor_reschedule	Update scheduling information to the central manager
condor_rm	Remove jobs from the queue
condor_run	Submit a shell command-line as a Condor job
condor_status	Display status of the Condor pool
condor_submit_dag	Manage and queue jobs within a specified DAG for inter-job dependencies.
condor_submit	Queue jobs for execution
condor_userlog	Display and summarize job statistics from job log files

Table 15.1
List of User Commands.

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

               Total                7               516                 5
```

Checking on the progress of jobs The condor_q command displays the status of all jobs in the queue. An example of the output from condor_q is

```
% condor_q

-- Schedd: uug.cs.wisc.edu : <128.115.121.12:33102>
 ID        OWNER            SUBMITTED     RUN_TIME ST PRI SIZE CMD
 55574.0   jane            6/23 11:33   4+03:35:28 R  0   25.7 seycplex seymour.d
 55575.0   jane            6/23 11:44   0+23:24:40 R  0   26.8 seycplexpseudo sey
 83193.0   jane            3/28 15:11  48+15:50:55 R  0   17.5 cplexmip test1.mp
 83196.0   jane            3/29 08:32  48+03:16:44 R  0   83.1 cplexmip test3.mps
 83212.0   jane            4/13 16:31  41+18:44:40 R  0   39.7 cplexmip test2.mps
```

```
 5 jobs; 0 idle, 5 running, 0 held
```

This output contains many columns of information about the queued jobs. The `ST` column (for status) shows the status of current jobs in the queue. An `R` in the status column means the the job is currently running. An `I` stands for idle. The status `H` is the hold state. In the hold state, the job will not be scheduled to run until it is released (via the `condor_release` command). The `RUN_TIME` time reported for a job is the time that job has been allocated to a machine as DAYS+HOURS+MINS+SECS.

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

```
000 (8135.000.000) 05/25 19:10:03 Job submitted from host: <128.105.146.14:1816>
...
001 (8135.000.000) 05/25 19:12:17 Job executing on host: <128.105.165.131:1026>
...
005 (8135.000.000) 05/25 19:13:06 Job terminated.
(1) Normal termination (return value 0)
Usr 0 00:00:37, Sys 0 00:00:00  -  Run Remote Usage
Usr 0 00:00:00, Sys 0 00:00:05  -  Run Local Usage
Usr 0 00:00:37, Sys 0 00:00:00  -  Total Remote Usage
Usr 0 00:00:00, Sys 0 00:00:05  -  Total Local Usage
9624  -  Run Bytes Sent By Job
7146159  -  Run Bytes Received By Job
9624  -  Total Bytes Sent By Job
7146159  -  Total Bytes Received By Job
...
```

The `condor_jobmonitor` tool parses the events in a user log file and can use the information to graphically display the progress of your jobs. Figure 15.2 contains a screenshot of `condor_jobmonitor` in action.

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

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

Name        Arch     OpSys        State      Activity   LoadAv Mem  ActvtyTime

alfred.cs. INTEL    LINUX        Claimed    Busy       0.980  64    0+07:10:02
```

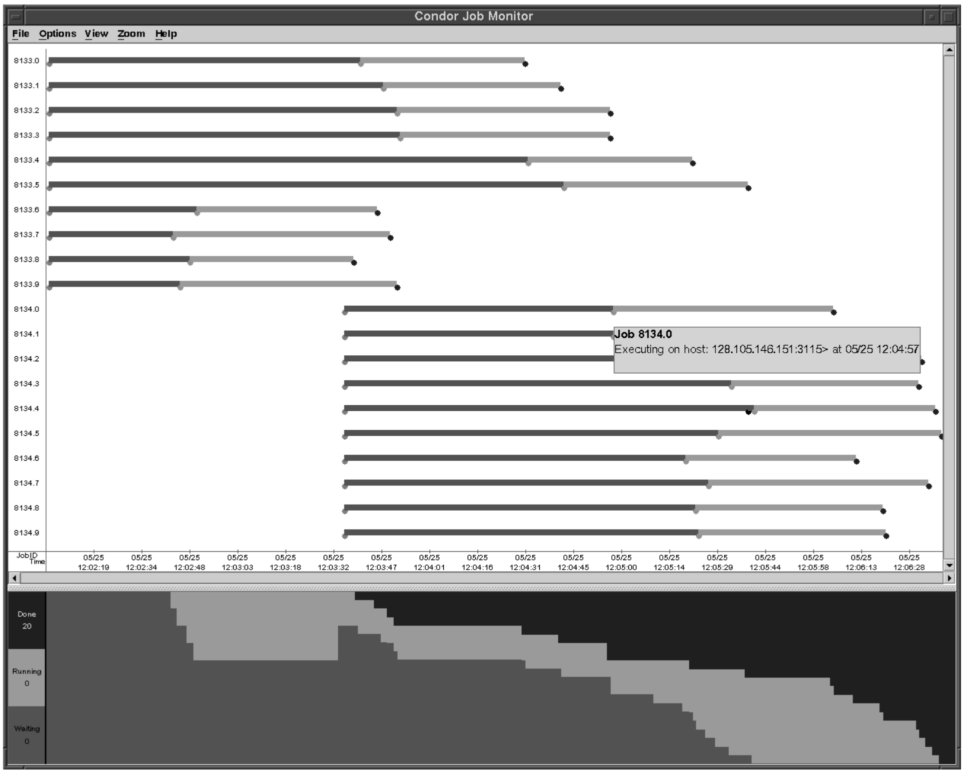

Figure 15.2
Condor JobMonitor Tool.

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

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

```
% condor_status -run

Name       Arch     OpSys       LoadAv RemoteUser           ClientMachine
```

```
adriana.cs INTEL    LINUX        0.980  hepcon@cs.wisc.edu   chevre.cs.wisc.
alfred.cs. INTEL    LINUX        0.980  breach@cs.wisc.edu   neufchatel.cs.w
amul.cs.wi INTEL    LINUX        1.000  nice-user.condor@cs. chevre.cs.wisc.
anfrom.cs. INTEL    LINUX        1.023  ashoks@jules.ncsa.ui jules.ncsa.uiuc
anthrax.cs INTEL    LINUX        0.285  hepcon@cs.wisc.edu   chevre.cs.wisc.
astro.cs.w INTEL    LINUX        1.000  nice-user.condor@cs. chevre.cs.wisc.
aura.cs.wi INTEL    LINUX        0.996  nice-user.condor@cs. chevre.cs.wisc.
balder.cs. INTEL    LINUX        1.000  nice-user.condor@cs. chevre.cs.wisc.
bamba.cs.w INTEL    LINUX        1.574  dmarino@cs.wisc.edu  riola.cs.wisc.e
bardolph.c INTEL    LINUX        1.000  nice-user.condor@cs. chevre.cs.wisc.
...
```

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

```
%  condor_q

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

2 jobs; 1 idle, 1 running, 0 held

%  condor_rm 132.0
Job 132.0 removed.

%  condor_q

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

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

Changing the priority of jobs In addition to the priorities assigned to each user, Condor provides users with the capability of assigning priorities to any submitted job. These job priorities are local to each queue and range from -20 to +20, with higher values meaning better priority.

The default priority of a job is 0. Job priorities can be modified using the `condor_prio` command. For example, to change the priority of a job to -15,

```
%  condor_q raman
```

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

1 jobs; 1 idle, 0 running, 0 held

%  condor_prio -p -15 126.0

%  condor_q raman

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

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

It is important to note that these *job* priorities are completely different from the *user* priorities assigned by Condor. Job priorities only control which one of *your* jobs should run next; it has no bearing on if your jobs will run before another user's jobs.

Why does the job not run? Users sometimes find that their jobs do not run. There are several reasons why a specific job does not run. These reasons include failed job or machine constraints, bias due to preferences, insufficient priority, and the preemption throttle that is implemented by the condor_negotiator to prevent thrashing. Many of these reasons can be diagnosed by using the *-analyze* option of condor_q. For example, the following job submitted by user jbasney had not run for several days.

```
% condor_q

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

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

Running condor_q's analyzer provided the following information:

```
%  condor_q 125.0 -analyze

-- Submitter: froth.cs.wisc.edu : <128.105.73.44:33847> : froth.cs.wisc.edu
---
125.000:  Run analysis summary.  Of 323 resource offers,
```

```
          323 do not satisfy the request's constraints
            0 resource offer constraints are not satisfied by this request
            0 are serving equal or higher priority customers
            0 are serving more preferred customers
            0 cannot preempt because preemption has been held
            0 are available to service your request

WARNING:  Be advised:
   No resources matched request's constraints
   Check the Requirements expression below:

Requirements = Arch == "INTEL" && OpSys == "IRIX6" &&
  Disk >= ExecutableSize && VirtualMemory >= ImageSize
```

The `Requirements` expression for this job specifies a platform that does not exist. Therefore, the expression always evaluates to FALSE.

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

If the problem persists and the analyzer is unable to detect the situation, the job may begin to run but immediately terminates and return to the idle state. Viewing the job's error and log files (specified in the submit command file) and Condor's `SHADOW_LOG` file may assist in tracking down the problem. If the cause is still unclear, please contact your system administrator.

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

By default, Condor will send you an email message when your job completes. You can modify this behavior with the `condor_submit` "notification" command. The message will include the exit status of your job or notification that your job terminated abnormally.

15.2.4 Submitting different types of jobs: Alternative Universes

A Universe in Condor defines an execution environment. Condor supports the following Universes on Linux:

- Vanilla
- MPI
- PVM
- Globus
- Scheduler
- Standard

The `Universe` attribute is specified in the submit description file. If the Universe is not specified, then it will default to Standard.

Vanilla Universe The Vanilla Universe is used to run serial (non-parallel) jobs. The examples provided in the previous section use the Vanilla Universe. Most Condor users prefer to use the Standard Universe to submit serial jobs because of several helpful features of the Standard Universe. However, the Standard Universe has several restrictions on the types of serial jobs supported. The Vanilla Universe, on the other hand, has no such restrictions. Any program that runs outside of Condor will run in the Vanilla Universe. Binary executables as well as scriptsare welcome in the Vanilla Universe.

A typical Vanilla Universe job relies on a shared filesystem between the submit machine and all the nodes in order to allow jobs to access their data. However, if a shared filesystem is not available, Condor can transfer the files needed by the job to and from the execute machine. See Section 15.2.5 for more details on this.

MPI Universe The MPI Universe allows parallel programs written with MPI to be managed by Condor. To submit an MPI program to Condor, specify the number of nodes to be used in the parallel job. Use the *machine_count* attribute in the submit description file, as in the example:

```
# Submit file for an MPI job which needs 8 large memory nodes
universe = mpi
executable = my-parallel-job
requirements = Memory >= 512
machine_count = 8
queue
```

Further options in the submit description file allow a variety of parameters, such as the job requirements or the executable to use across the different nodes.

At the publication time of this book, Condor expects your MPI job to be linked with the MPICH implementation of MPI configured with the `ch_p4` device (see Chapter ??, specifically Section ??). However, support for different devices and MPI implementations are expected, so check the documentation included with your specific version of Condor for additional information on how your job should be linked with MPI for Condor.

If your Condor pool consists of both dedicated compute machines (that is, Beowulf cluster nodes) and opportunistic machines (that is, desktop workstations), realize that by default Condor will schedule MPI jobs to run on the dedicated resources only.

PVM Universe The PVM Universe allows master-worker style parallel programs written for the Parallel Virtual Machine interface (see Chapter ??) to be used with Condor. A unique aspect of the PVM Universe is that PVM jobs submitted to Condor can harness both dedicated and non-dedicated (opportunistic) workstations throughout the pool by dynamically adding machines to and removing machines from the parallel virtual machine as machines become available.

In the PVM Universe, Condor acts as the resource manager for the PVM daemon. Whenever a PVM program asks for nodes via a `pvm_addhosts()` call, the request is forwarded to Condor. Condor finds a machine in the Condor pool using ClassAd matching mechanisms, and adds it to the virtual machine. If a machine needs to leave the pool, the PVM program is notified by normal PVM mechanisms, for example, the `pvm_notify()` call.

There are several different parallel programming paradigms. One of the more common is the "master-worker" or "pool of tasks" arrangement. In a master-worker program model, one node acts as the controlling master for the parallel application and sends pieces work out to worker nodes. The worker node does some computation and sends the result back to the master node. The master has a pool of work that needs to be done, and it assigns the next piece of work out to the next worker that becomes available.

The PVM Universe is designed to run PVM applications which follow the master-worker paradigm. Condor runs the master application on the machine where the job was submitted and will not preempt the master application. Workers are pulled in from the Condor pool as they become available.

Writing a PVM program that deals with Condor's opportunistic environment can be a tricky task. For that reason, the MW framework has been created. MW is a tool for making master-worker style applications in Condor's PVM Universe.

For more information, see the MW Homepage online at `http://www.cs.wisc.edu/condor/mw`.

Submitting to the PVM Universe is similar to submitting to the MPI Universe, except the syntax for `machine_count` is different to reflect the dynamic nature of the PVM Universe. Here is a simple sample submit description file:

```
# Require Condor to give us one node before starting
# the job, but we'll use up to 75 nodes if they are
# available.
universe = pvm
executable = master.exe
machine_count = 1..75
queue
```

By using `machine_count = <min>..<max>`, the submit description file tells Condor that before the PVM master is started, there should be at least `<min>` number of machines given to the job. It also asks Condor to give it as many as `<max>` machines.

More detailed information on the PVM Universe is available in the Condor Manual as well as on the Condor-PVM Homepage at URL `http://www.cs.wisc.edu/condor/pvm`.

Globus Universe The Globus Universe in Condor is intended to provide the standard Condor interface to users who wish to submit jobs to machines being managed by Globus (`http://www.globus.org`).

Scheduler Universe The Scheduler Universe is used to submit a job which will immediately run on the *submit* machine, as opposed to a remote execution machine. The purpose is to provide a facility for job *meta-schedulers* that desire to manage the submission and removal of jobs into a Condor queue. Condor includes one such meta-scheduler which utilizes the Scheduler Universe: the DAGMan scheduler, which can be used to specify complex interdependencies between jobs. See Section 15.2.6 for more on DAGMan.

Standard Universe The Standard Universe requires minimal extra effort on the part of the user, but provides a serial job with the following highly desireable services:

- Transparent process *checkpoint* and *restart*
- Transparent process migration
- Remote System Calls
- Configurable file I/O buffering
- On-the-fly file compression/inflation

Process Checkpointing in the Standard Universe A checkpoint of an executing program is a snapshot of the program's current state. It provides a a way for the program to be continued from that state at a later time. Using checkpoints gives Condor the freedom to reconsider scheduling decisions through preemptive-resume scheduling. If the scheduler decides to rescind a machine that is running a Condor job (for example, when the owner of that machine returns and reclaims it, or when a higher priority user desires the same machine), the scheduler can take a checkpoint of the job and preempt the job without losing the work the job has already accomplished. The job can then be resumed later when the Condor scheduler allocates it a new machine. Additionally, periodic checkpoints provide fault tolerance. Normally, when performing long-running computations, if a machine crashes, or must be rebooted for an administrative task, all the work that has been done is lost. The job must be restarted from the beginning, which can mean days, weeks, or even months of wasted computation time. With checkpoints, Condor ensures that positive progress is always made on jobs, and that only the computation done since the last checkpoint is lost. Condor can be take checkponts periodically, and after an interruption in service, the program can continue from the most recent snapshot.

To enable taking checkpoints, no changes are required to a program's source code. The program must be relinked with the Condor system call library (see below). Taking the checkpoint of a process is implemented in the Condor system call library as a signal handler. When Condor sends a checkpoint signal to a process linked with this library, the provided signal handler writes the state of the process out to a file or a network socket. This state includes the contents of the process's stack and data segments, all CPU state (including register values), the state of all open files, and any signal handlers and pending signals. When a jobs is to be continued using a checkpoint, Condor reads this state from the file or network socket, restoring the stack, shared library and data segments, file state, signal handlers, and pending signals. The checkpoint signal handler then restores the CPU state and returns to the user code, which continues from where it left off when the checkpoint signal arrived. Condor jobs submitted to the Standard Universe will automatically perform a checkpoint when preempted from a machine. When a suitable replacement execution machine is found (of the same architecture and operating system), the process is restored on this new machine from the checkpoint, and computation is resumed from where it left off.

By default, a checkpoint is written to a file on the local disk of the submit machine. A Condor checkpoint server is also available to serve as a repository for checkpoints.

Remote System Calls in the Standard Universe One hurdle to overcome when placing an job on a remote execution workstation is data access. In order to utilize the remote resources, the job must be able to read from and write to files on its submit machine. A requirement that the remote execution machine be able to access these files via NFS, AFS, or any other network file system may significantly limit the number of eligible workstations and therefore hinder the ability of a HTC environment to achieve high throughput. Therefore, in order to maximize throughput, Condor strives to be able to run any application on any remote workstation of a given platform without relying upon a common administrative setup. The enabling technology that permits this is Condor's Remote System Calls mechanism. This mechanism provides the benefit that Condor does not require a user to posses a login account on the execute workstation.

When a Unix process needs to access a file, it calls a file I/O system function such as open(), read(), or write(). These functions are typically handled by the standard C library, which consists primarily of stubs that generate a corresponding system call to the local kernel. Condor users link their applications with an enhanced standard C library via the `condor_compile` command. This library does not duplicate any code in the standard C library; instead, it augments certain system call stubs (such as the ones which handle file I/O) into remote system call stubs. The remote system call stubs package the system call number and arguments into a message which is sent over the network to a `condor_shadow` process that runs on the submit machine. Whenever Condor starts a Standard Universe job, it also starts a corresponding shadow process on the initiating host where the user originally submitted the job (see Figure 15.3). This shadow process acts as an agent for the remotely executing program in performing system calls. The shadow then executes the system call on behalf of the remotely running job in the normal way. The shadow packages up the results of the system call in a message and sends it back to the remote system call stub in the Condor library on the remote machine. The remote system call stub returns its result to the calling procedure, which is unaware that the call was done remotely rather than locally. In this fashion, calls in the user's program to open(), read(), write(), close(), and all other file I/O calls transparently take place on the machine which submitted the job instead of the remote execution machine.

Relinking and Submitting for the Standard Universe To convert a program into a Standard Universe job, use the `condor_compile` command to re-link with the Condor libraries. Place `condor_compile` in front of your usual link command. You do not need to modify the program's source code, but you do need access

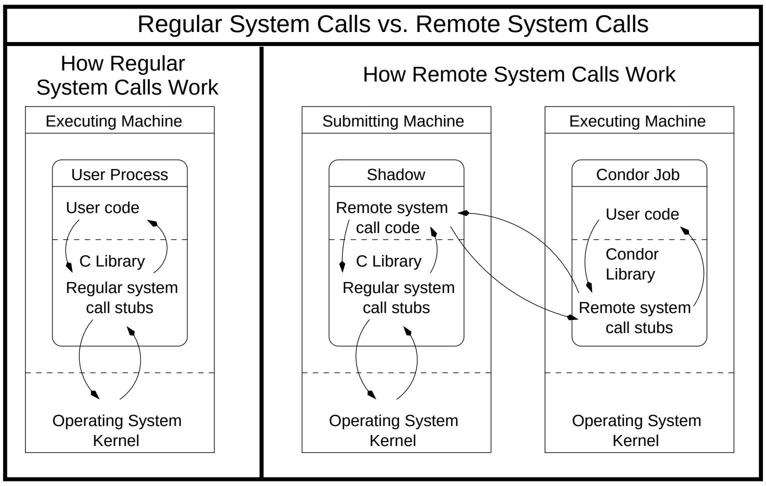

Figure 15.3
Remote System Calls in the Standard Universe.

to its un-linked object files. A commercial program that is packaged as a single executable file cannot be converted into a Standard Universe job.

For example, if you normally link your job by executing:

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

You can re-link your job for Condor with:

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

After you have re-linked your job, you can submit it. A submit description file for the Standard Universe is similar to one for the Vanilla Universe. However, several additional submit directives are available to perform activities such as on-the-fly compression of data files. Here is an example:

```
# Submit 100 runs of my-program to the Standard Universe
universe = standard
executable = my-program.exe
# Each run should take place in a seperate subdirectory: run0, run1, ...
initialdir = run$(Process)
# Ask the Condor remote syscall layer to automatically compress
# on-the-fly any writes done by my-program.exe to file data.output
compress_files = data.output
queue 100
```

Many system administrator's prefer Condor's checkpoint implementation at the user level instead of kernel-level alternatives, for fear that pervasive kernel changes could compromise the stability of the operating system.

Standard Universe Limitations Condor performs its process checkpoint and migration routines strictly in user mode – there are no kernel drivers with Condor. Because Condor is not operating at the kernel level, there are limitations on what process state it is able to checkpoint. As a result, the following restrictions are imposed upon Standard Universe jobs:

1. Multi-process jobs are not allowed. This includes system calls such as *fork()*, *exec()*, and *system()*.

2. Interprocess communication is not allowed. This includes pipes, semaphores, and shared memory.

3. Network communication must be brief. A job *may* make network connections using system calls such as *socket()*, but a network connection left open for long periods will delay checkpoints and migration.

4. Multiple kernel-level threads are not allowed. However, multiple user-level threads (green threads) *are* allowed.

5. All files should be accessed read-only or write-only. A file which is both read and written to could cause trouble if a job must be rolled back to an old checkpoint image.

6. On Linux, your job must be statically linked. Dynamic linking is allowed in the Standard Universe on some other platforms supported by Condor, and perhaps this restriction on Linux will be removed in a future Condor release.

15.2.5 Giving your job access to its data files

Once your job starts on a machine in your pool, how does it access its data files? Condor provides alternatives.

If the job is a Standard Universe job, then Condor solves the problem of data access automatically using the Remote System call mechanism described above. Whenever the job tries to open, read, or write to a file, the I/O will actually take place on the submit machine, whether or not a shared file system is in place.

Condor can utilize a shared file system, if one is available and permanently mounted across the machines in the pool. This is usually the case in a Beowulf cluster. But what if your Condor pool includes non-dedicated (desktop) machines as well? You could specify a `Requirements` expression in your submit description file to require that jobs only run on machines which actually do have access to a common, shared file system. Or, you could request in the submit description file that Condor transfer your job's data files using the Condor File Transfer mechanism.

When Condor finds a machine willing to execute your job, it can create a temporary subdirectory for your job on the execute machine. The Condor File Transfer mechanism will then send via TCP the job executable(s) and input files from the submitting machine into this temporary directory on the execute machine. After the input files have been transferred, the execute machine will start running the job with the temporary directory as the job's current working directory. When the job completes or is kicked off, Condor File Transfer will automatically send back to the submit machine any output files created or modified by the job. After the files have been sent back successfully, the temporary working directory on the execute machine is deleted.

Condor's File Transfer mechanism has several features to ensure data integrity in a non-dedicated environment. For instance, transfers of multiple files are performed atomically.

Condor File Transfer behavior is specified at job submission time using the submit description file and `condor_submit`. Along with all the other job submit description parameters, you can use the following File Transfer commands in the submit description file:

transfer_input_files = < file1, file2, file... > Use this parameter to list all the files which should be transferred into the working directory for the job before the job is started.

transfer_output_files = < file1, file2, file... > Use this parameter to explicitly list which output files to transfer back from the temporary working directory on the execute machine to the submit machine. Most of the time, however, there is no need to use this parameter. If `transfer_output_files` is not specified, Condor will automatically transfer back all files in the job's temporary working directory which have been modified or created by the job.

transfer_files = <ONEXIT — ALWAYS — NEVER> Setting `transfer_files` equal to `ONEXIT` will cause Condor to transfer the job's output files back to the submitting machine only when the job completes (exits). Specifying `ALWAYS` tells Condor to transfer back the output files when the job completes *or* when Condor kicks off the job (preempts) from a machine prior to job completion. The `ALWAYS` option is specifically intended for fault-tolerant jobs which periodocially write out their state to disk and can restart where the left off. Any output files transferred back to the submit machine when Condor preempts a job will automatically be sent back out again as input files when the job restarts.

15.2.6 The DAGMan scheduler

The DAGMan scheduler within Condor allows the specification of dependencies between a set of programs. A directed acyclic graph (DAG) can be used to represent a set of programs where the input, output, or execution of one or more programs is dependent on one or more other programs. The programs are nodes (vertices) in the graph, and the edges (arcs) identify the dependencies. Each program within the DAG becomes a job submitted to Condor. The DAGMan scheduler enforces the depencies of the DAG.

An input file to DAGMan identifies the nodes of the graph, as well as how to submit each job (node) to Condor. It also specifies the graph's dependencies and describes any extra processing that comprises a node of the graph, but must take place just before or just after the job is run.

A simple diamond-shaped DAG with four nodes is given in Figure **??**.

A simple input file to DAGMan for this diamond-shaped DAG may be

```
# file name: diamond.dag
Job  A  A.condor
Job  B  B.condor
Job  C  C.condor
Job  D  D.condor
PARENT A CHILD B C
PARENT B C CHILD D
```

The four nodes are named `A`, `B`, `C`, and `D`. Lines beginning with the keyword `Job` identify each node by giving it a name, and they also specify a file to be used as a submit description file for submission as a Condor job. Lines with the keyword `PARENT` identify the dependencies of the graph. Just like regular Condor submit description files, lines with a leading pound character ('#') are comments.

The DAGMan scheduler uses the graph to order the submission of jobs to Condor. The submission of a child node will not take place until the parent node has successfully completed. There is no ordering of siblings imposed by the graph, and therefore DAGMan does not impose an ordering when submitting the jobs to Condor. For the diamond-shaped example, nodes `B` and `C` will be submitted to Condor in parallel.

Each job within the example graph uses a different submit description file. An example submit description file for job `A` may be

```
# file name: A.condor
executable   = nodeA.exe
```

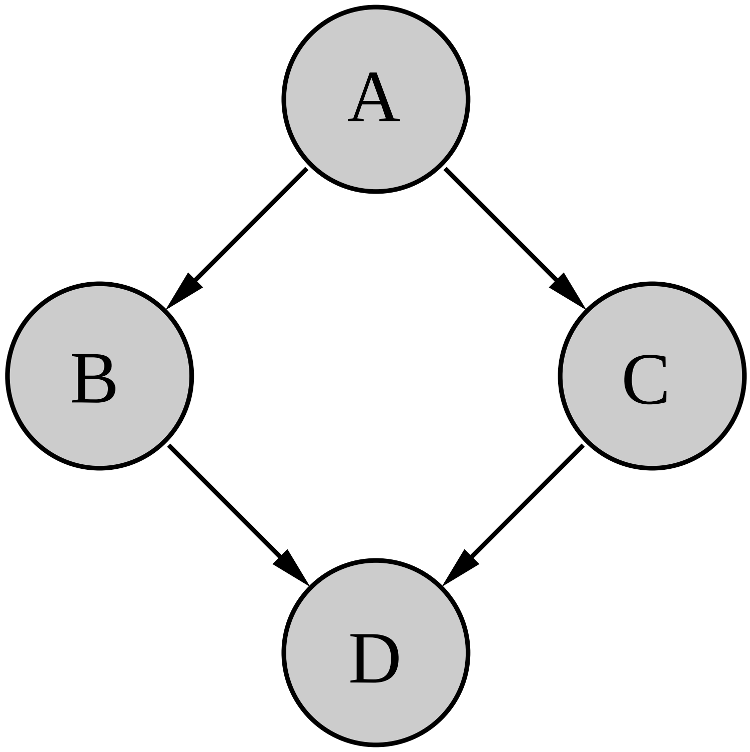

Figure 15.4
A directed acyclic graph with four nodes.

```
output      = A.out
error       = A.err
log         = diamond.log
universe    = vanilla
queue
```

An important restriction for submit description files of a DAG is that each node of the graph must use the same log file. DAGMan uses the log file in the enforcement of the graph's dependencies.

Submission of the graph for execution under Condor uses the Condor tool `condor_submit_dag`. For the diamond-shaped example, submission would use the command:

```
condor_submit_dag diamond.dag
```

15.3 Condor Architecture

A Condor pool is comprised of a single machine which serves as the *central manager*, and an arbitrary number of other machines that have joined the pool. Conceptually, the pool is a collection of resources (machines) and resource requests (jobs). The role of Condor is to match waiting requests with available resources. Every part of Condor sends periodic updates to the central manager, the centralized repository of information about the state of the pool. The central manager periodically assesses the current state of the pool and tries to match pending requests with the appropriate resources.

15.3.1 The Condor Daemons

To help understand the architecture of Condor, the following list describes all the daemons (background server processes) in Condor and what role they play in the system:

`condor_master` This daemon's role is to simplify system administration. It is responsible for keeping the rest of the Condor daemons running on each machine in a pool. The master spawns the other daemons and periodically checks the timestamps on the binaries of the daemons it is managing. If it finds new binaries, the master will restart the affected daemons. This allows Condor to be upgraded easily. In addition, if any other Condor daemon on the machine exits abnormally, the `condor_master` will send e-mail to the system administrator with information about the problem and then automatically restart the affected daemon. The `condor_master` also supports various administrative commands to start, stop or reconfigure daemons remotely. The `condor_master` runs on every machine in your Condor pool.

`condor_startd` This daemon represents a machine to the Condor pool. It advertises a machine ClassAd which contains attributes about the machine's capabilities and policies. Running the startd enables a machine to execute jobs. The `condor_startd` is responsible for enforcing the policy under which remote jobs will be started, suspended, resumed, vacated, or killed. When the startd is ready to execute a Condor job, it spawns the `condor_starter`, described below.

`condor_starter` This program is the entity that spawns the remote Condor job on a given machine. It sets up the execution environment and monitors the job once it is running. The starter detects job completion, sends back status information to the submitting machine, and exits.

`condor_schedd` This daemon represents jobs to the Condor pool. Any machine that allows users to submit jobs needs to have a `condor_schedd` running. Users submit jobs to the schedd, where they are stored in the *job queue*. The various tools to view and manipulate the job queue (such as `condor_submit`, `condor_q`, or `condor_rm`) connect to the schedd to do their work.

`condor_shadow` This program runs on the machine where a job was submitted whenever that job is executing. The shadow serves requests for files to transfer, logs the job's progress, and reports statistics when the job completes. Jobs that are linked for Condor's Standard Universe, which perform remote system calls, do so via the `condor_shadow`. Any system call performed on the remote execute machine is sent over the network to the `condor_shadow`. The shadow performs the system call (such as file I/O) on the submit machine and the result is sent back over the network to the remote job.

`condor_collector` This daemon is responsible for collecting all the information about the status of a Condor pool. All other daemons periodically send ClassAd updates to the collector. These ClassAds contain all the information about the state of the daemons, the resources they represent or resource requests in the pool (such as jobs that have been submitted to a given schedd). The `condor_collector` can be thought of as a dynamic database of ClassAds. The `condor_status` command can be used to query the collector for specific information about various parts of Condor. The Condor daemons also query the collector for important information, such as what address to use for sending commands to a remote machine. The `condor_collector` runs on the machine designated as the central manager.

`condor_negotiator` This daemon is responsible for all the matchmaking within the Condor system. The negotiator is also responsible for enforcing user priorities in the system.

15.3.2 The Condor Daemons in Action

Within a given Condor installation, one machine will serve as the pool's central manager. In addition to the `condor_master` daemon which runs on every machine in a Condor pool, the central manager runs the `condor_collector` and the `condor_negotiator` daemons. Any machine in the installation that should be capable of running jobs should run the `condor_startd`, and any machine which should maintain a job queue and therefore allow users on that machine to submit jobs should run a `condor_schedd`.

Condor allows any machine to simultaneously execute jobs and serve as a submission point by running both a `condor_startd` and a `condor_schedd`. Figure 15.5 displays a Condor pool where every machine in the pool can both submit and run jobs, including the central manager.

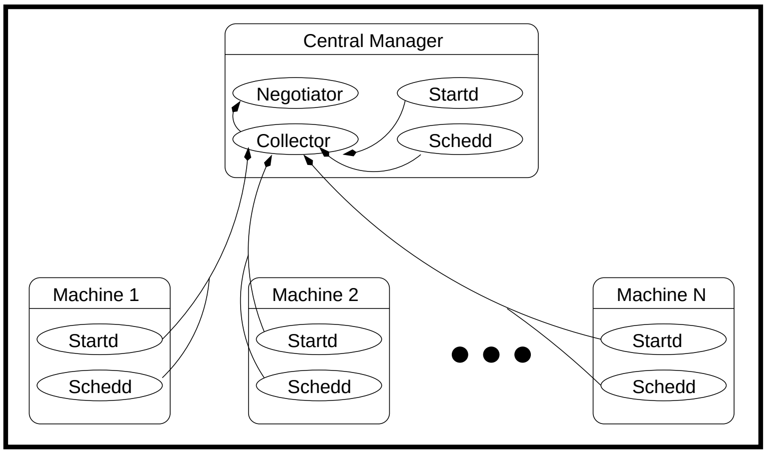

Figure 15.5
Daemon layout of an idle Condor pool.

The interface for adding a job to the Condor System is `condor_submit`, which reads a job description file, creates a job ClassAd, and gives that ClassAd to the `condor_schedd` managing the local job queue. This triggers a *negotation cycle.* During a negotiation cycle, the `condor_negotiator` queries the `condor_collector` to discover all machines that are willing to perform work and all users with idle jobs. The `condor_negotiator` communicates *in user priority order* with each `condor_schedd` that has idle jobs in its queue, and performs matchmaking to match jobs with machines such that both job and machine ClassAd requirements are satisfied and preferences (rank) are honored.

Once the `condor_negotiator` makes a match, the `condor_schedd` claims the corresponding machine and is allowed to make subsequent scheduling decisions about the order in which jobs run. This hierarchical, distributed scheduling architecture enhances Condor's scalability and flexibility.

When the `condor_schedd` starts a job, it spawns a `condor_shadow` process on the submit machine and the `condor_startd` spawns a `condor_starter` process on the corresponding execute machine (see Figure 15.6). The shadow transfers the job ClassAd and any data files required to the starter, which spawns the user's application.

If the job is a Standard Universe job, the shadow will begin to service remote system calls originating from the user job, allowing the job to transparently access data files on the submitting host.

When the job completes or is aborted, the `condor_starter` removes every process spawned by the user job, and frees any temporary scratch disk space used by the job. This ensures that the execute machine is left in a clean state, and resources (such as processes or disk space) are not being leaked.

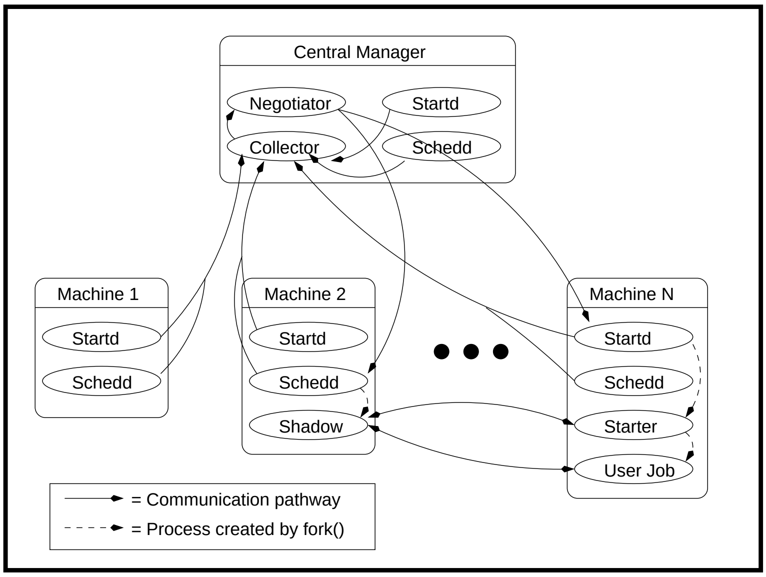

Figure 15.6
Daemon layout when a job submitted from Machine 2 is running.

15.4 Installing Condor under Linux

The first step towards the installation of Condor is to download the software from the Condor website at `http://www.cs.wisc.edu/condor/downloads`. There is no cost to download or use Condor.

On the website you will find complete documentation and release notes for the different versions and platforms we support. Please take care to download the appropriate version of Condor for your platform (the operating system and processor architecture).

Before you begin the installation, there are several issues you need to decide and actions to perform.

Creation of user condor. For both security and performance reasons, the Condor daemons should execute with root privileges. However, to avoid running as root except when absolutely necessary, the Condor daemons will run with the privileges of user condor on your system. In addition, the condor user simplifies installation, since files owned by the user condor will be created, and the home directory of the user condor can be used to specify file locations. For Linux clusters, we highly recommend that you create the user condor on all machines before installation begins.

Location. Administration of your pool is eased when the release directory, which includes all the binaries, libraries, and configuration files used by Condor, is placed on a shared file server. Note that one set of binaries is needed for each platform in your pool.

Administrator. Condor needs an e-mail address for an administrator. Should Condor need assistance, this is where e-mail will be sent.

Central manager. The central manager of a Condor pool does matchmaking and collects information for the pool. Choose a central manager that has a good network connection and is likely to be online all the time (or at least rebooted quickly in the event of a failure).

Once you have decided the answers to these questions (and set up the condor user) you are ready to begin installation. The tool called `condor_install` is executed to begin the installation. The configuration tool will ask you a short series of questions, mostly related to the issues addressed above. Answer the questions appropriately for your site, and Condor will be installed.

On a large Linux cluster, you can speed up the installation process by running `condor_install` once on your fileserver node, and configure your entire pool at the same time. If you use this configuration option, you will only need to run the `condor_init` script (which requires no input) on each of your compute nodes.

The default Condor installation will configure your pool to assume non-dedicated resources. Section 15.5 will discuss how to configure and customize your pool for a dedicated cluster.

However, after installation, there are a few security configuration settings you will want to customize right away. Condor implements security at the host (or machine) level. A set of configuration defaults set by the installation deal with access to the Condor pool by host. Given the distributed nature of the daemons that implement Condor, access to these daemons is naturally host-based. Each daemon can be given the ability to allow or deny service (by host) within its configuration. Within the access levels available, *Read*, *Write*, *Administrator*, and *Config* are important to set correctly for each pool of machines.

Read Allows a machine to obtain information from Condor. Examples of information that may be read are the status of the pool and the contents of the job queue.

Write Allows a machine to provide information to Condor, such as submit a job or join the pool.

Administrator Allows a user on the machine to effect privileged operations such as changing a user's priority level, or starting and stopping the Condor system from running.

Config Allows a user on the machine to change Condor's configuration settings remotely using the `condor_config_val` tool's *-set* and *-rset* options. This has very serious security implications, so we recommend you do not enable Config access to any hosts.

The defaults during installation give all machines read and write access. The central manager is also given administrator access.

You will probably want to change these defaults for your site. Please read the Condor Administrator's Manual for details on network authorization in Condor, and how to customize it for your wishes.

15.5 Configuring Condor

This section describes how to configure and customize Condor for your site. It discusses the configuration files used by Condor, how to configure the policy for starting and stopping jobs in your pool, and provides recommended settings for using Condor on a cluster.

There are a number of configuration files that facilitate different levels of control over how Condor is configured on each machine in a pool. The top-level or global configuration file is shared by all machines in the pool. For ease of administration, this file should be located on a shared file system. In addition, there may be multiple local configuration files for each machine, allowing the local settings to override the global settings. This allows each machine to potentially have different daemons running, different policies for when to start and stop Condor jobs, and so on.

All of Condor's configuration files should be owned and only writable by root. It is very important to maintain strict control over these files, since they contain security sensitive settings.

15.5.1 Location of Condor's Configuration Files

Condor has a default set of locations it uses to try to find its top-level configuration file. The locations are checked in the following order:

1. The file specified in the `CONDOR_CONFIG` environment variable.

2. '`/etc/condor/condor_config`', if it exists.

3. If user condor exists on your system, the '`condor_config`' file in this user's home directory.

If a Condor daemon or tool cannot find its global configuration file when it starts up, it will print out an error message and immediately exit. However, once the global configuration file has been read by Condor, any other local configuration files can be specified with the `LOCAL_CONFIG_FILE` macro.

This macro can contain a single entry if you only want two levels of configuration (global and local). If you need a more complex division of configuration values (for example, if you have machines of different platforms in the same pool and desire separate files for platform-specific settings), `LOCAL_CONFIG_FILE` can contain a list of files.

Condor provides other macros to help you easily define the location of the local configuration files for each machine in your pool. Most of these are special macros with evaluate to different values depending on which host is reading the global configuration file.

- `HOSTNAME` : The hostname of the local host.
- `FULL_HOSTNAME` : The fully-qualified hostname of the local host.
- `TILDE` : The home directory of the condor user on the local host.
- `OPSYS` : The operating system of the local host, for example: "LINUX", "WINNT4" (for Windows NT), or "WINNT5" (for Windows 2000). This is primarily useful in heterogeneous clusters with multiple platforms.

- `RELEASE_DIR` : The directory where Condor is installed on each host. This macro is defined in the global configuration file, and is set by Condor's installation program.

By default, the local configuration file is defined as:

```
LOCAL_CONFIG_FILE = $(TILDE)/condor_config.local
```

15.5.2 Recommended Configuration File Layout for a Cluster

The ease of administration is an important consideration in a cluster, particularly if you have a large number of nodes. To make Condor easy to configure, we highly recommend that you install all of your Condor configuration files, even the per-node local configuration files, on a shared filesystem. That way, you can easily make changes in one place.

You should use a subdirectory in your release directory for holding all of the local configuration files. By default, Condor's release directory contains an '`etc`' directory for this purpose.

You should create separate files for each node in your cluster, using the hostname as the first half of the filename, and ".local" as the end. For example, if your cluster nodes are named "n01", "n02" and so on, the files should be called '`n01.local`', '`n02.local`', etc. These files should all be placed in your '`etc`'directory, described above.

In your global configuration file, you would use the following setting to describe the location of your local configuration files:

```
LOCAL_CONFIG_FILE = $(RELEASE_DIR)/etc/$(HOSTNAME).local
```

The central manager of your pool needs special settings in its local configuration file. These attributes are set automatically by the Condor installation program. The rest of the local configuration files can be left empty at first.

Having your configuration files laid out in this way will help you more easily customize Condor's behavior on your cluster. We will discuss other possible config scenarios at the end of this chapter.

NOTE: We recommend that you store all of your Condor configuration files under a version control system, such as CVS. While this is not required, it will help you keep track of the changes you make to your configuration, who made them, when they occurred, and why. In general, it is a good idea to store configuration files under a version control system, since none of the above concerns are specific to Condor.

15.5.3 Customizing Condor's Policy Expressions

Certain configuration expressions are used to control Condor's policy for executing, suspending, and evicting jobs. Their interaction can be somewhat complex. Defining an inappropriate policy impacts the throughput of your cluster and the happiness of its users. If you are interested in creating a specialized policy for your pool, we recommend that you read the Condor Administrator's Manual. Only a basic introduction follows.

All policy expressions are ClassAd expressions and are defined in Condor's configuration files. Policies are usually pool-wide and are therefore defined in the global configuration file. However, if individual nodes in your pool require their own policy, the appropriate expressions can be placed in local configuration files.

The policy expressions are treated by the `condor_startd` as part of its machine ClassAd (along with all the attributes you can view with `condor_status -long`). They are always evaluated against a job ClassAd, either by the `condor_negotiator` when trying to find a match, or by the `condor_startd` when it is deciding what to do with the job that is currently running. Therefore, all policy expressions can reference attributes of a job, such as the memory usage or owner, in addition to attributes of the machine, such as keyboard idle time or CPU load.

Most policy expressions are ClassAd boolean expressions, so they evaluate to either TRUE, FALSE, or UNDEFINED. UNDEFINED occurs when an expression references a ClassAd attribute that is not found in either the machine's ClassAd or the ClassAd of the job under consideration. For some expressions, this is treated as a fatal error, so you should be sure to use the ClassAd *meta-operators*, described in section 15.1.2 when referring to attributes which might not be present in all ClassAds.

An explanation of policy expressions requires the understanding of the different stages that a job can go through from initially executing until the job completes or is evicted from the machine. Each policy expression is then described in terms of the step in the progression that it controls.

The Lifespan of a Job Executing in Condor When a job is submitted to Condor, the `condor_negotiator` performs matchmaking to find a suitable resource to use for the computation. This process involves satisfying both the job and the machine's requirements for each other. The machine can define the exact conditions under which it is willing to be considered available for running jobs. The job can define exactly what kind of machine it is willing to use.

Once a job has been matched with a given machine, there are four states the job can be in: running, suspended, graceful shutdown, and quick shutdown. As soon as the match is made, the job sets up its execution environment and begins running.

While it is executing, a job can be suspended (for example, due to other activity on the machine where it is running). Once it has been suspended, the job can either resume execution, or can move on to preemption or eviction.

All Condor jobs have two methods for preemption: graceful and quick. Standard universe jobs are given a chance to produce a checkpoint with graceful preemption. For the other universes, graceful implies that the program is told to get off the system, but it is given time to clean up after itself. On all flavors of Unix, a SIGTERM is sent during graceful shutdown by default, although users can override this default when they submit their job. A quick shutdown involves rapidly killing all processes associated with a job, without giving them any time to execute their own clean up procedures. The Condor system performs checks to ensure that processes are not left behind once a job is evicted from a given node.

Condor Policy Expressions This section describes the various expressions used to control the policy for starting, suspending, resuming, and preempting jobs.

`START` When the `condor_startd` is willing to start executing a job.

`RANK` How much the `condor_startd` prefers each type of job running on it. The `RANK` expression is a floating point value, instead of a boolean value. The `condor_startd` will preempt the job it is currently running if there is another job in the system that yields a higher value for this expression.

`WANT_SUSPEND` Controls if the `condor_startd` should even consider suspending this job or not. In effect, it determines which expression, `SUSPEND` or `PREEMPT`, should be evaluated while the job is running. `WANT_SUSPEND` does not control when the job is actually suspended; use the `SUSPEND` expression.

`SUSPEND` When the `condor_startd` should suspend the currently running job. If `WANT_SUSPEND` evaluates to TRUE, `SUSPEND` is periodically evaluated whenever a job is executing on a machine. If `SUSPEND` becomes TRUE, the job will be suspended.

`CONTINUE` If and when the `condor_startd` should resume a suspended job. The `CONTINUE` expression is only evaluated while a job is suspended. If it evaluates to TRUE, the job will be resumed and the `condor_startd` will go back to the Claimed/Busy state.

`PREEMPT` When the `condor_startd` should preempt the currently running job. This expression is evaluated whenever a job has been suspended. If `WANT_SUSPEND` evaluates to FALSE, `PREEMPT` is checked while the job is executing.

`WANT_VACATE` If Condor is preempting a job (because the `PREEMPT` expression evaluates to TRUE), `WANT_VACATE` determines if the job should be evicted gracefully or quickly. If `WANT_VACATE` is FALSE, the `condor_startd` will immediately kill the job and all of its child processes whenever it must evict the application. If `WANT_VACATE` is TRUE, the `condor_startd` performs a graceful shutdown, instead.

`KILL` When the `condor_startd` should give up on a graceful preemption and move directly to the quick shutdown.

`PREEMPTION_REQUIREMENTS` This expression is used by the `condor_negotiator` when it is performing match-making, not by the `condor_startd`. While trying to schedule jobs on resources in your pool, the `condor_negotiator` considers the priorities of the various users in the system (see section 15.6.3 for more details). If a user with a better priority has jobs waiting in the queue and no resources are currently idle, the matchmaker will consider preempting another user's jobs and giving those resources to the user with the better priority. This process is known as *priority preemption*. The `PREEMPTION_REQUIREMENTS` expression must evaluate to TRUE for such a preemption to take place.

`PREEMPTION_RANK` This floating point value is evaluated by the `condor_negotiator`. If the matchmaker decides it must preempt a job due to user priorities, the `PREEMPTION_RANK` macro determines which resource to preempt. Among the set of all resources that make the `PREEMPTION_REQUIREMENTS` expression evaluate to TRUE, the one with the highest value for `PREEMPTION_RANK` is evicted.

15.5.4 Customizing Condor's Other Configuration Settings

In addition to the policy expressions, there are other settings you will need to modify to customize Condor for your cluster.

`DAEMON_LIST` The comma-separated list of daemons that should be spawned by the `condor_master`. As described in section 15.3.1 discussing the architecture of Condor, each host in your pool can play different roles depending upon which daemons are started on it. You define these roles using the `DAEMON_LIST` in the appropriate configuration files to enable or disable the various Condor daemons on each host.

DedicatedScheduler The name of the dedicated scheduler for your cluster. This setting must have the form:

```
DedicatedScheduler = "DedicatedScheduler@full.host.name.here"
```

15.6 Administration Tools

Condor has a rich set of tools for the administrator. Figure 15.2 gives an overview of the Condor commands typically used solely by the system administrator. Of course, many of the "user-level" Condor tools summarized in Figure 15.1 can be very helpful for cluster administration as well. For instance, the condor_status tool can easily display the status for all nodes in the cluster, including dynamic information such as current load average and free virtual memory.

Command	Description
condor_checkpoint	Checkpoint jobs running on the specified hosts
condor_config_val	Query or set a given Condor configuration variable
condor_master_off	Shut down Condor and the condor_master
condor_off	Shut down Condor daemons
condor_on	Start up Condor daemons
condor_reconfig	Reconfigure Condor daemons
condor_restart	Restart the condor_master
condor_stats	Display historical information about the Condor pool
condor_userprio	Display and manage user priorities
condor_vacate	Vacate jobs that are running on the specified hosts

Table 15.2
Commands reserved for the administrator.

15.6.1 Remote Configuration and Control

All machines in a Condor pool can be remotely managed from a centralized location. Condor can be enabled, disabled, or restarted remotely using the condor_on, condor_off, and condor_restart commands respectively. Additionally, any aspect of Condor's configuration file on a node can be queried or changed remotely via the condor_config_val command. Of course, not anyone is allowed to change your Condor configuration remotely. Doing so requires proper authorization, which is set up at installation time (see Section 15.4).

Many aspects of Condor's configuration, including its scheduling policy, can be changed on-the-fly without requiring the pool to be shutdown and restarted. This is accomplished using the condor_reconfig command which asks the Condor daemons on a specified host to re-read the Condor configuration files and take appropriate action– on the fly if possible.

15.6.2 Accounting and Logging

Condor keeps many statistics about what is happening in the pool. Each daemon can be asked to keep a detailed log of its activities; Condor will automatically rotate these log files when they reach a maximum size as specified by the administrator.

In addition to the condor_history command, which allows users to view job ClassAds for jobs which have previously completed, the condor_stats tool can be used to query for historical usage statistics from a pool-wide accounting database. This database contains information about how many jobs were being serviced for each user at regular intervals, as well as how many machines were busy. For instance, condor_stats could be asked to display the total number of jobs running at 5 minute intervals for a specified user between January 15th and January 30th.

The condor_view tool takes the raw information obtainable with condor_stats and converts it into HTML, complete with interactive charts. Figure 15.7 shows a sample display of the output from condor_view in a web browser. Using condor_view, the site administrator can quickly put detailed, real-time usage statistics about the Condor pool onto a web site.

15.6.3 User Priorities in Condor

The job queues in Condor are not strictly FIFO (First In, First Out). Instead, Condor implements *priority queueing*. Different users will get different sized allocations of machines depending upon their current user priority, regardless of how many jobs are from a competing user are "ahead" of them in the queue. Condor can also be configured to perform *priorty preemption* if desired. For instance, suppose user A is using all the nodes in a cluster, when suddenly a user with a superior priority submits jobs. With priority preemption enabled, Condor will preempt the jobs of the lower priority user in order to immediately start the jobs submitted by the higher priority user.

Starvation of the lower priority users is prevented by a fair-share algorithm, which attempts to give all users the same amount of machine allocation time over a specified interval. In addition, the priority calculations in Condor are *based on ratios* instead of absolutes. For example, if Bill has a priority which is twice as good as Fred, Condor will not starve user Fred by allocating all machines to Bill.

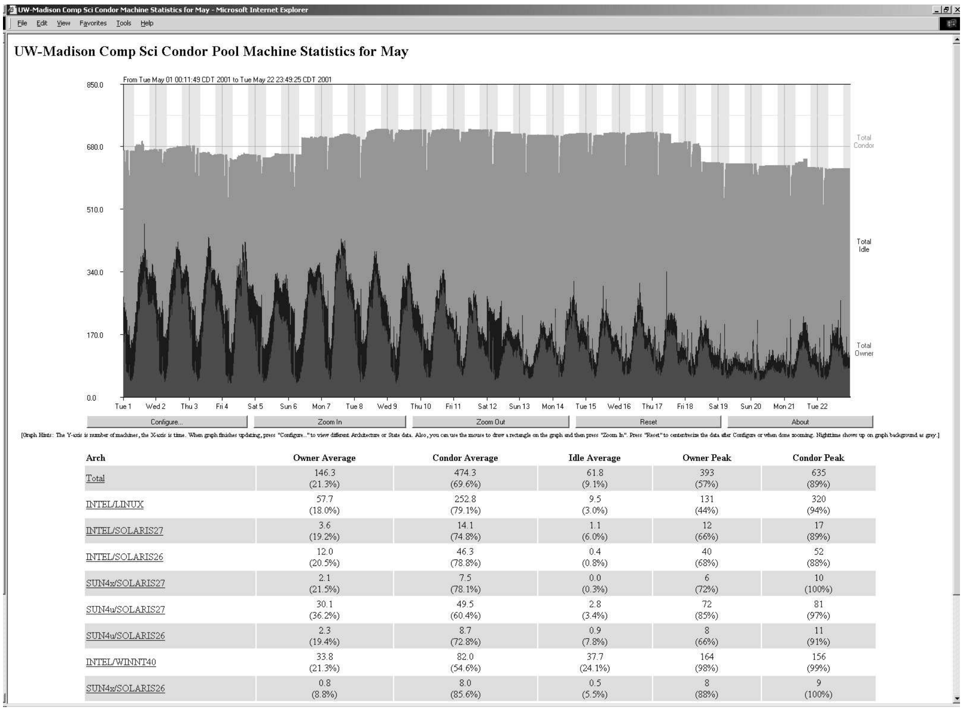

Figure 15.7
CondorView displaying machine usage.

Instead, Bill will get, on average, twice as many machines as user Fred because Bill's priority is twice as good.

The `condor_userprio` command can be used by the administrator to view or edit a user's priority. `condor_userprio` can also be used to override Condor's default fair-share policy and explicitly assign users a better or worse priority in relation to other users.

15.7 Cluster Setup Scenarios

This section explores different scenarios for how to configure your cluster. Five scenarios are presented, along with a basic idea of what configuration settings you will need to modify, or what steps you will need to take for each scenario.

1. A uniformly owned, dedicated compute cluster, with a single front-end node for submission, and support for MPI applications.

2. A cluster of multi-processor nodes.

3. A cluster of distributively owned nodes. Each node prefers to run jobs submitted by its owner.

4. Desktop submission to the cluster.

5. Expanding the cluster to non-dedicated (desktop) computing resources.

Most of these scenarios can be combined. Each scenario builds on the previous one to add further functionality to the basic cluster configuration.

15.7.1 Basic Configuration: Uniformly Owned Cluster

The most basic scenario involves a cluster where all resources are owned by a single entity, and all compute nodes enforce the same policy for starting and stopping jobs. All compute nodes are dedicated, meaning that they will always start an idle job and they will never preempt or suspend until completion. There is a single front-end node for submitting jobs, and dedicated MPI jobs are enabled from this host.

To enable this basic policy, your global configuration file must contain these settings:

```
START = True
SUSPEND = False
CONTINUE = False
PREEMPT = False
KILL = False
WANT_SUSPEND = True
WANT_VACATE = True
RANK = Scheduler =?= $(DedicatedScheduler)
DAEMON_LIST = MASTER, STARTD
```

The final entry listed here specifies that the default role for nodes in your pool is execute-only.

The `DAEMON_LIST` on your front-end node must also enable the `condor_schedd`. This front-end node's local configuration file will be:

```
DAEMON_LIST = MASTER, STARTD, SCHEDD
```

15.7.2 Using Multi-Processor Compute Nodes

If any node in your Condor pool is a symmetric multiprocessor (SMP) machine, Condor will represent that node as multiple virtual machines (VMs), one for each CPU. By default, each VM will have a single CPU and an even share of all shared system resources, such as RAM and swap space. If this behavior satisfies your needs, you do not need to make any configuration changes for SMP nodes to work properly with Condor.

Some sites might want different behavior of their SMP nodes. For example, assume your cluster was composed of dual-processor machines with 1 gigabyte of RAM, and one of your users was submitting jobs with a memory footprint of 700 megabytes. With the default setting, all VMs in your pool would only have 500 megabytes of RAM, and your user's jobs would never run.

In this case, you would want to unevenly divide RAM between the two CPUs, to give half of your VMs 750 megabytes of RAM. The other half of the VMs would be left with 250 megabytes of RAM.

There is more than one way to divide shared resources on an SMP machine with Condor, all of which are discussed in detail in the Condor Administrator's Manual. The most basic method is described here.

To unevenly divide shared resources on an SMP, you must define different *virtual machine types*, and tell the `condor_startd` how many virtual machines of each type to advertise. The most simple method to define a virtual machine type is to specify what fraction of all shared resources each type should receive.

For example, if you wanted to divide a 2-node machine where one CPU received one quarter of the shared resources, and the other CPU received the other three quarters, you would use the following settings:

```
VIRTUAL_MACHINE_TYPE_1 = 1/4
VIRTUAL_MACHINE_TYPE_2 = 3/4
NUM_VIRTUAL_MACHINES_TYPE_1 = 1
NUM_VIRTUAL_MACHINES_TYPE_2 = 1
```

If you only want to unevenly divide certain resources, and split the rest evenly, you can specify separate fractions for each shared resource. This is described in detail in the Condor Administrator's Manual.

15.7.3 Scheduling a Distributively Owned Cluster

Many clusters are owned by more than one entity. Two or more smaller groups might pool their resources to buy a single, larger cluster. In these situations, it is important that the group which paid for a portion of the nodes should get priority to run on those nodes.

Each resource in a Condor pool can define its own `Rank` expression, which specifies the kinds of jobs it would prefer to execute.

If a cluster is owned by multiple entities, you can divide the cluster's nodes up into groups, based on ownership. Each node would set `Rank` such that jobs coming from the group that owned it would have the highest priority.

Assume there is a 60-node compute cluster at a university, shared by three departments: Astronomy, Math, and Physics. Each department contributed the funds for 20 nodes. Each group of 20 nodes would define its own `Rank` expression. The Astronomy department's settings:

```
Rank = Department == "Astronomy"
```

The users from each department would also add a `Department` attribute to all of their job ClassAds. The administrators could configure Condor to automatically add this attribute to all job ads from each site (see the Condor Administrator's Manual for details).

If the entire cluster was idle and a Physics user submitted 40 jobs, she would see all 40 of her jobs start running.

However, if a user in Math submitted 60 jobs and a user in Astronomy submitted 20 jobs, 20 of the Physicist's jobs' would be preempted, and each group would get 20 machines out of the cluster.

If all of the Astronomy department's jobs completed, the Astronomy nodes would go back to serving Math and Physics jobs. The Astronomy nodes would continue to run Math or Physics jobs until either some Astronomy jobs were submitted, or all the jobs in the system completed.

15.7.4 Submitting to the Cluster from Desktop Workstations

Most organizations that install a compute cluster have other workstations at their site. It is usually desirable to allow these machines to act as front-end nodes for the cluster, so users can submit their jobs from their own machines and have the applications execute on the cluster. Even if there is no shared file system between the cluster and the rest of the computers, Condor's remote system calls and file transfer functionality can enable jobs to migrate between the two and still access their data (see section 15.2.5 on accessing data files for details).

To enable a machine to submit into your cluster, run the Condor installation program and specify that you want to setup a *submit-only* node. This will set the `DAEMON_LIST` on the new node to be:

```
DAEMON_LIST = MASTER, SCHEDD
```

The installation program will also create all the directories and files needed by Condor.

NOTE: You can only have one node configured as the dedicated scheduler for your pool. Do not attempt to add a second submit node for MPI jobs.

15.7.5 Expanding the Cluster to Non-Dedicated (Desktop) Computing Resources

One of the most powerful features in Condor is the ability to combine dedicated and *opportunistic* scheduling within a single system. Opportunistic scheduling involves placing jobs on non-dedicated resources under the assumption that the resources might not be available for the entire duration of the jobs. Opportunistic scheduling is used for all jobs in Condor with the exception of dedicated MPI applications.

If your site has a combination of jobs and uses applications other than MPI, you should strongly consider adding all of your computing resources, even desktop workstations, to your Condor pool. With /iflinux checkpointing and process migration, /fi suspend and resume capabilities, opportunistic scheduling and matchmaking, Condor can harness the idle CPU cycles of any machine and put them to good use.

To add other computing resources to your pool, run the Condor installation program and specify that you want to configure a node that can both submit and execute jobs. The default installation sets up a node with a policy for starting, suspending, and preempting jobs based on the activity of the machine (for example, keyboard idle time and CPU load). These nodes will not run dedicated MPI jobs, but they will run jobs from any other universe, including PVM.

15.8 Conclusion

Condor is a powerful tool for scheduling jobs across platforms, both within and beyond the boundaries of your Beowulf clusters. Through its unique combination of both dedicated and opportunistic scheduling, Condor provides a unified framework for high throughput computing.

This chapter was processed by LaTeX on August 23, 2001.