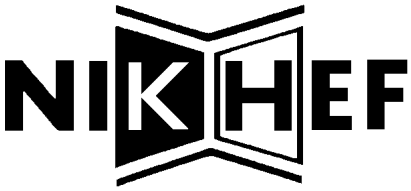
Condor Flocking: Load Sharing between Pools of Workstations

X. Evers

Supervision:

Delft University of Technology
Department of Mathematics and Computing
Operating Systems and Distributed Systems Group
P.O. Box 356, 2600 AJ Delft, The Netherlands

Prof. Dr. I.S. Herschberg
Dr. Ir. D.H.J. Epema
Ir. J.F.C.M. de Jongh
Drs. J.W.J. Heijnsdijk



National Institute for Nuclear Physics and High-Energy Physics
P.O. Box 41882, 1009 DB Amsterdam, The Netherlands

Dr. R. van Dantzig (SMC)

April 1993

Contents

Preface

This report is the result of a six-month graduating term for the Operating Systems and Distributed Systems group of the Technical University of Delft. This assignment was performed within the Computer System Group of NIKHEF (National Institute for Nuclear Physics and High-Energy Physics) under the supervision of R. van Dantzig (Spin Muon Collaboration), and I.S. Herschberg, D.H.J. Epema and J.F.C.M. de Jongh (Technical University of Delft).

I would like to thank Leen Dikken, Frank van der Linden, Peter Sloot, Peter Trenning and Joep Vesseur for the interest they showed in my work, and Peter Trenning also for helping me with the installation of the Condor flock at the FWI.

I would like to thank Ronald Boontje for his advise and help, Ronald taught me a lot about UNIX and networks, and I would like to thank the other members of the CSG for providing a stimulating atmosphere and wonderful work environment.

Miron Livny guided me in the right direction during a visit to the NIKEF in November 1992, and I'm grateful to Mike Litzkow and Maarten Litmaath for answering numerous questions on the Condor system.

I would like to thank Dick Epema who provided me with this assignment and, together with Jan de Jongh, supervised my graduating term for the TU-Delft. I also want to give special thanks to Rene van Dantzig for his encouragement and endless enthusiasm throughout my stay at NIKHEF.

Last, but not least, my parents and my fiancée, Florence, gave me lots of emotional support, without which I could never have succeeded.

Xander Evers, Amsterdam, April 1993.

Chapter 1

Introduction

Condor is a facility for executing UNIX jobs on a pool of cooperating workstations. Jobs are queued and executed remotely on workstations at times when those workstations would otherwise be idle. A transparent checkpointing mechanism is provided, and jobs migrate from workstation to workstation without user intervention. Condor is meant for long-running, computation-intensive jobs that require no user interaction. Examples of this kind of jobs are simulations and combinatoric searches. Condor was developed by the Computer Science department of the university of Wisconsin - Madison. The Condor system is described in Chapter 2. Some of the data structures and algorithms used internally by the Condor control software are described in Chapter 3.

A Condor flock is a collection of Condor pools that share the load of Condor jobs in some way. For instance, the flock may consist of the Condor pools of institutes working together on a project, departments of a company, or faculties of a university. We make a distinction between the situation where one user has the right to run his jobs in several pools, and the situation where the owners of a number of Condor pools have reached agreement to share the load of jobs between their Condor pools. In the first situation, the user will want to distribute his jobs over the pools in such a way, that the total turn-around time is minimal. In the second situation, the objective depends on the kind of agreement reached between the owners of the Condor pools. In most cases the objective will be to decrease the wait-while-idle time, that is, to decrease the time that Condor jobs are waiting while there are idle machines in the flock on which these jobs could execute. Another distinction we make is based on whether or not it is possible to use the remote execution mechanism of Condor in the flock situation.

My assignment consisted of designing and implementing a first version of a Condor flock, for the situation where the owners of the pools have decided to couple their pools, and where it is acceptable to use the remote executing mechanism of Condor. In Chapter 4 we discuss some general design decisions for this situation. These decisions form together the ideal design for this situation as we see it.

Chapter 5 describes the design of the Condor flock such as it has implemented during my six month graduating term. A very important restriction posed to the design was that as

little as possible should be changed to the existing Condor software. The design presented in Chapter 5 differs therefore from the ideal design. Chapter 6 describes how this system has been implemented. The Condor flock has been implemented by adding a virtual machine to each Condor pool, called the World Machine. The World Machines of a Condor flock work together to run Condor jobs of one pool on free machines of other pools.

The results of some performance tests concerning remote execution across wide area networks are presented in Chapter 7. In Chapter 8 the possible directions of future development are indicated. The source code of the World Machine is provided in the appendix.

Chapter 2

Overview of the Condor System

Many organizations own hundreds of powerful workstations which are connected by local area networks. These workstations are often allocated to a single user who exercises full control over the workstation's resources. Litzkow and Livny [13] stated that in such an environment you can find three types of users, *casual* users who seldom utilize the full capacity of their machines, *sporadic* users who for short periods of time fully utilize the capacity of the workstation they own, and *frustrated* users who for long periods of time have computing demands that are beyond the power of their workstations. The throughput of these frustrated users is limited by the power of their workstations. In a paper by Mutka and Livny [16] it was shown that in a computing environment of workstations connected by a local area network, about 70% of the workstations is available for remote execution.

Condor is a software package for executing long-running, computation-intensive jobs on workstations that would otherwise be idle. Condor was designed to meet the challenge posed by the frustrated users, namely to provide convenient access to unutilized workstations while preserving the rights of their owners. Condor has been developed by the Computer Science department of the university of Wisconsin - Madison. This chapter gives a summary of the documentation [2] [3] [11] [12] [13] [14] [15] on Condor.

2.1 Design Features

Several principles have driven the design of Condor.

- Workstation owners should always have the resources of the workstation they own at their disposal. Workstation owners are generally happy to let somebody else compute on their machines while they are out, but they want their machines back promptly upon returning, and they don't want to have to take special action to regain control. Immediate response is the reason most people prefer a dedicated workstation over access to a time sharing system. Condor handles this automatically.

- Remote capacity should be easy to access. The Condor software is responsible for locating and allocating idle workstations. Condor users do not have to search for idle

machines. The local execution environment is preserved for remotely executing processes. Users do not have to worry about moving data files to remote workstations before executing programs there. Users of Condor may be assured that their jobs will eventually complete, because jobs are periodically checkpointed. If a user submits a job to Condor which runs on somebody else's workstation, but the job is not finished when the workstations owner returns, the job will be restarted from the latest checkpoint as soon as possible on another machine.

- No special programming should be required to use Condor. Condor is able to run normal UNIX programs, only requiring the user to relink, not to recompile them or change any code. Condor does its work completely outside the kernel, and is compatible with Berkeley 4.2 and 4.3 UNIX kernels and many of their derivatives. Because it requires no changes to the operating system, Condor is portable and can be used in environments where access to the internals of the system is not possible. Condor does pay a price for this flexibility in both the speed and completeness of its process migration [14].

2.2 Remote System Calls

The user program is provided with the illusion that it is operating in the environment of the initiating machine. In some circumstances file I/O is redirected from the machine where execution actually takes place to the initiating machine. In other situations files on the initiating machine are accessed more efficiently by use of NFS.

Every UNIX program, whether or not written in the C language, is linked with the C library. In the normal situation this library provides the interface between the user program and the UNIX kernel. This interface is implemented as stubs, which perform the system call on behalf of the user program. Figure 2.1 illustrates the normal UNIX system call mechanism.

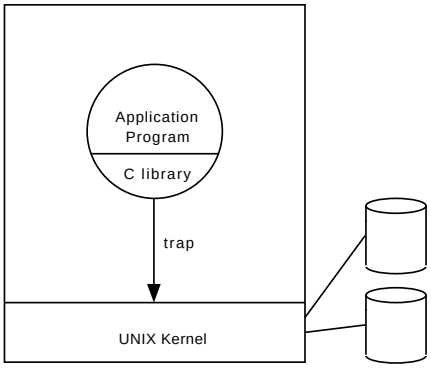


Figure 2.1 Normal UNIX system calls.

Figure 2.2 shows how the system call mechanism has been altered by providing a special version of the C library which performs system calls remotely. This special library, like the normal library, has a stub for each UNIX system call. These stubs either execute a request locally by mimicking the normal stubs or pack the request into a message which is sent to the

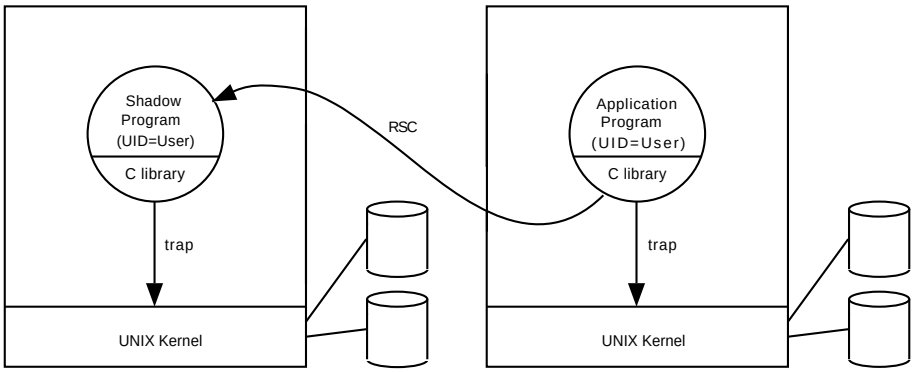


Figure 2.2 Remote system calls.

Shadow process. The Shadow executes the system call on the initiating machine, packs the results, and sents them back to the stub. The stub then returns to the application program in exactly the same way the normal system call would have, had the call been local. The Shadow runs with the same user and group ids, and in the same directory as the user process would have had it been executing on the initiating machine.

If a network file system such as NFS is in use, an important optimization is possible. Performance can be increased in these cases by avoiding remote system calls, and accessing the file directly. This mechanism works as follows. At the time of an `open` request, the stub sends a name translation request to the initiating machine. The Shadow process responds with a translated pathname in the form of *hostname:pathname*, where *hostname* may refer to a file server, and the *pathname* is the name by which the file is known on the server (which may be different from the pathname on the initiating machine, because of mount points and symbolic links). The stub then examines the mount table on the machine where it is executing, and if possible accesses the file without using the Shadow process. Whenever a process is checkpointed and restarted on another machine, the name translation process is repeated, since access to remotely mounted files may vary among the execution machines.

2.3 Checkpointing

Condor provides a transparent checkpointing mechanism which allows it to take a checkpoint of a running job, and migrate that job to another workstation when the machine it is currently running on becomes busy with non-Condor activity. This allows Condor to return workstations to their owners promptly, yet provide assurance to Condor users that their jobs will make progress, and eventually complete.

Ideally, checkpointing and restarting a process means storing the process state and later restoring it in such a way that the process can continue where it left off. In the most general case, the state of a UNIX process may include pieces of information which are known only to the kernel, or which may not be possible to recreate. Condor is meant for jobs whose state is simple enough that they can be checkpointed. In the current version of Condor only single process jobs are supported. This means that the *fork(2)*, *exec(2)*, and similar calls

are not implemented. Signals and signal handlers (the *signal(3)*, *sigvec(2)*, *kill(2)* calls), and interprocess communication (IPC) calls (*socket(2)*, *send(2)*, *recv(2)*, etc) are not supported. The state of a UNIX process includes the contents of memory (the text, data and stack segments), processor registers and the status of open files. The approach of the designers of Condor to saving and restoring the state of a process is to rely on basic UNIX mechanisms to keep Condor easy to port. The checkpoint file is itself a UNIX executable file.

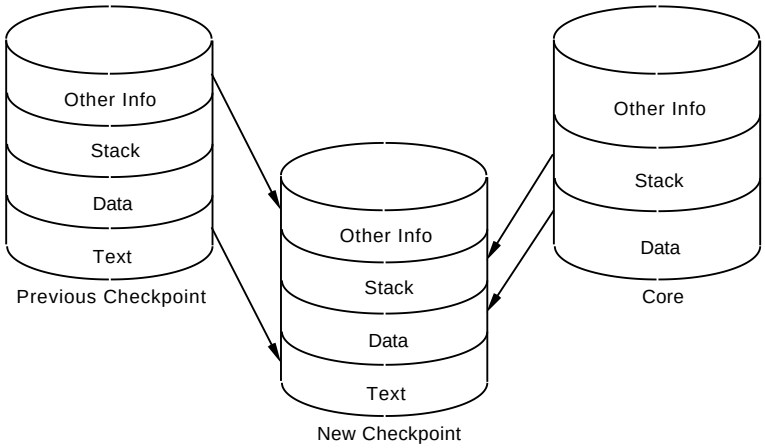


Figure 2.3 Creating a checkpoint file.

Figure 2.3 shows that a new checkpoint file is created from pieces of the previous checkpoint and a core image. The text segment of the new checkpoint is an exact copy of the text segment of the old checkpoint. The data area and stack area are copied from the core file into the new checkpoint file. The setjmp/longjmp facilities of the C library have been used to save the register contents and program counter. Information about currently open files is gathered by the stubs of the *open*, *close* and *dup* system calls, of Condor's special C library. The exact way in which Condor makes a checkpoint file can be found in a paper by Litzkow and Solomon [14]. Before a Condor process is executed for the first time, its executable file is modified to look exactly like a checkpoint file, so that every checkpoint is done in the same way.

2.4 Control Software

The Condor control software consists of two daemons which run on each member of the Condor pool, the *Schedd* and the *Startd*, and two daemons which run on a single machine called the *Central Manager Machine*, the *Collector* and the *Negotiator*. The Collector and the Negotiator are separate processes, but they can be viewed as one logical process called the Central Manager. An additional daemon, the *Kbdd*, is necessary on machines running the X window system.

The Condor daemons are started and (if necessary) restarted by the *Condor_Master*. This daemon starts those Condor daemons that are appropriate for the machine it is running on. If

one of the daemons it is monitoring dies, the Condor_Master will attempt to restart it. It will also send mail to the Condor administrator describing the problem. The Condor_Master limits its restart attempts to a certain number per hour. If this limit is exceeded, the Condor_Master will abort. The Condor_Master should also be started by */etc/rc* or */etc/rc.local* so that it will be restarted in the event of a crash.
The Condor daemons have the following tasks:

- The **Schedd** maintains the queue of Condor jobs that have been submitted from the machine on which the Schedd is running. The Schedd has the responsibility of prioritizing the jobs in the queue. The Schedd will periodically send a message to the Collector to give information about its job queue.
- The **Startd** determines whether the machine on which it runs, is idle, and is responsible for starting and managing the foreign job if one is running on its machine. The Startd periodically informs the Collector about the state of its machine.
- The **Kbdd** informs the Startd about the keyboard and mouse "idle time".
- The **Collector** collects information about the state of the Condor pool. The Collector is informed periodically by the Startd and Schedd of each machine on whether the machine is available or not, and how many jobs that machine wants to run.
- The **Negotiator** is responsible for allocating the idle machines to other machines which have Condor jobs to run. Periodically, the Negotiator asks information about the state of the pool from the Collector. It updates the priorities of the machines, does the scheduling, and returns the updated priorities to the Collector.

To illustrate how the daemons work together, we will follow a Condor job from the moment of submitting to the moment that it finishes or the owner of the hosting workstation returns. Figure 2.4 illustrates the situation when there are no Condor jobs running.

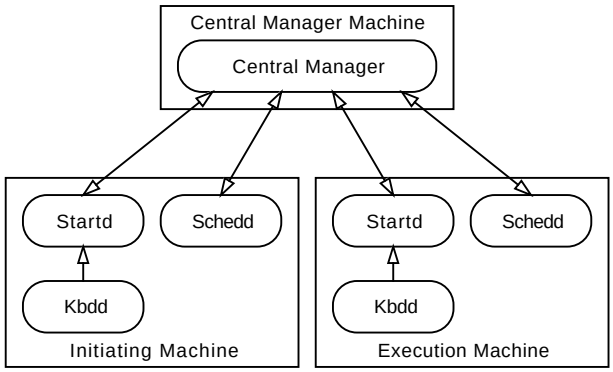


Figure 2.4 Condor daemons with no jobs running.

The Central Manager keeps track of which machines are idle, and which machines have Condor jobs to run. Periodically, the Central Manager will allocate the idle machines to machines that

want to run jobs. The Central Manager will contact the Sched-daemons of machines that want to run jobs, and determines for these jobs if they can run on one of the idle machines. The Central Manager will give permission to the *initiating* machine to run a job on the *execution* machine. The Schedd on the initiating machine spawns off a *Shadow* process to serve the job. The Shadow will then contact the Startd on the execution machine to ask if the machine is still idle. If the situation on the execution machine hasn't changed since the last update to the Central Manager, the execution machine will still be idle, and will respond with an OK. The Startd on the execution machine then spawns a process called the *Starter*. The Starter is responsible to start and manage the remotely running job (Figure 2.5).

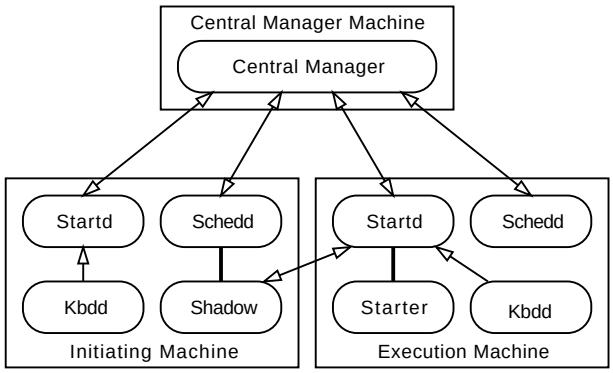


Figure 2.5 Condor processes while starting a job.

The Shadow on the initiating machine will transfer the checkpoint file to the Starter on the execution machine. The Starter spawns off the remote job. The Starter is responsible to give the user job periodically a "checkpoint" signal, causing the user job to save its file state and stack, and then to make a core dump. A new checkpoint file is made by the Starter and stored temporarily on the execution machine. The Starter restarts the job from the new version of the checkpoint and sets a timer for the next time it has to give the user job a checkpoint signal. The Shadow process on the initiating machine will handle the system calls for the user job (Figure 2.6).

If the user job finishes, the Starter and Shadow clean up, and the user is notified by mail that the job has finished. If the owner of the execution machines returns, the Startd on the execution machine will detect this, and it will send a "suspend" signal to the Starter, which will temporarily suspend the user job. This is because frequently the owners of machines are active for only a few seconds, then become idle again. This would be the case if the owner was just checking if there was new mail for example. If the execution machine remains busy for a certain period, the Startd will send a "vacate" signal to the Starter, who will abort the user job and return the latest checkpoint file to the Shadow on the initiating machine. If the user job had not run long enough to reach a checkpoint, the job is just aborted. No new checkpoint is made when the owner returns, because making a checkpoint is an I/O intensive activity and it should be avoided that the returned owner notices any interference from Condor.

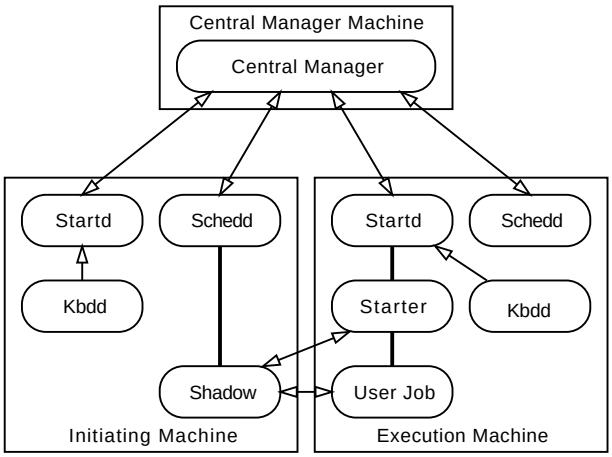


Figure 2.6 Condor processes with one job running.

2.5 The Condor Configuration Files

Condor can be customized by means of the *Condor configuration files*. There is one generic configuration file with definitions for all machines in a Condor pool, and for every machine a local configuration file. A definition in the local configuration file overrules a definition in the generic configuration file. In this way the generic configuration file can be used to globally configure a Condor pool, and the local configuration file can be used to make changes for individual machines. The generic configuration file and the local configuration files together will be called the Condor configuration files.

There are two sorts of definitions in the Condor configuration files, *macros* and *control functions*. Macros provide string-valued constants which do not change throughout the life of a daemon. Examples of these are names of log files and action intervals. Control functions provide arithmetic, boolean, or string-valued expressions which can be evaluated dynamically at run time. Examples of control functions are functions to update priorities and functions that determine when a machine should host a job. The rest of this chapter describes the working of a Condor system configured with the original config files, as distributed by the University of Wisconsin—Madison.

2.6 The Scheduling Algorithm

Condor uses a two-level scheduling algorithm to allocate idle machines to Condor jobs. On each machine the Schedd maintains the queue of Condor jobs that have been submitted from the machine on which the Schedd is running. The Schedd must prioritize its own jobs. The Central Manager does not keep track of individual jobs on the member machines. Instead it keeps track of how many jobs a machine wants to run, and how many it is running at any particular time. The Central Manager has the job of prioritizing the machines which want to run jobs.

The following assumptions were made:

- All users are entitled to equal rights.
- A workstation is owned by one or more users, which will always submit their jobs on this workstation. This means that the words user(s) and workstation are used in the same context.

The Central Manager uses the *up-down* algorithm to prioritize the machines. The up-down scheduling algorithm was presented in a paper by Mutka and Livny [15]. The goal of the up-down algorithm is to give all users a fair share of available remote processing cycles. Fair allocation is achieved by trading off the amount of execution time already allocated to a user and the amount of time the user has waited for an allocation. The up-down algorithm tries to protect the rights of light users when a few heavy users try to monopolize all free machines, without degrading throughput.

Every machine (i.e. user) in the Condor pool has a priority to run jobs. Initially all the machines have the priority 0. Periodically (normally every 300 seconds), the Negotiator updates the priorities of the machines in the Condor pool, and negotiates with the machines that want to run jobs. The control functions used by the Negotiator to update the machine priorities are:

```
INACTIVE    : Users <= 0
UPDATE_PRIO : Prio + Users - Running
```

"Prio" is the previous priority of the machine, "Users" is the number of different users that have Condor jobs in the queue, and "Running" is the number of Condor jobs the machine has currently running. If a machine is active (Users > 0), the control function UPDATE_PRIO is used to update the priority of this machine. If a machine is inactive (Users ≤ 0), the priority will be incremented by one if the priority is negative, and decremented by one if the priority is positive. The result is that machines which are running lots of jobs will tend to have low priorities, and machines which have jobs to run, but can't run them, will accumulate high priorities.

The Schedd is responsible for prioritizing its own jobs. The following control function is used by the Schedd to assign priorities to its jobs.

```
PRIO : (UserPrio * 10) + Expanded - (Qdate / 1000000000)
```

"UserPrio" is defined by the job owner in a similar (but opposite) way as the UNIX "nice" value. These values range from -20 to +20, with higher numbers corresponding to greater priority. "Expanded" will be 1 if the job has already done some execution. This is done to preserve disk space, an expanded job is bigger than an unexpanded job. "Qdate" is the UNIX time (seconds since 1-1-1970) the job was submitted. The constants make that "UserPrio" is the major criteria, "Expanded" is less important and "Qdate" is the minor criteria in determining job priority.

2.7 Machine States

The standard interval at which the Startd checks if its state should be changed is 5 seconds. Every 120 seconds the Negotiator is informed by the Startd about the machine status. The state of a machine is NOJOB when the keyboard has been idle for 15 minutes and the load average is below 0.3. A Condor job will be suspended if the load average becomes higher than 1.5 or when the keyboard is touched. The maximum time a Condor job can be suspended is 10 minutes. A Condor job will be resumed if the load average is below 0.3 and the keyboard has not been touched for 5 minutes. If a Condor job is not resumed within 10 minutes, the job will be vacated (the checkpoint file will be moved from the execution host to the initiating host). If a job has not been vacated within 10 minutes, the Condor job is killed. All of the above mentioned intervals can be changed to meet a workstation owner's wishes. Figure 2.7 shows all the different states a machine can be in.

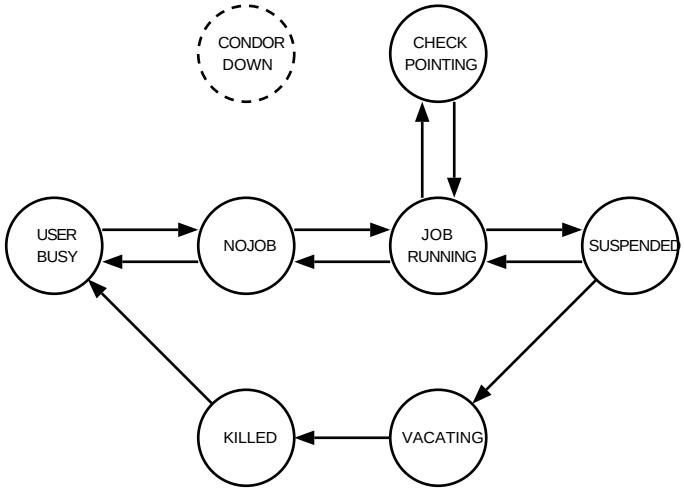


Figure 2.7 States of Condor machines.

The Starter is responsible for giving the Condor job a checkpoint signal at the appropriate time. In the condor configuration files the minimum and maximum interval is defined at which the Starter should give a checkpoint signal. Normally the first checkpoint is made after 30 minutes. After the first checkpoint the interval is doubled, until it is as big as the maximum interval (normally 2 hours). This is done, because the expectation of the period that a given machine will stay idle increases as the period that this machine is already idle increases.

2.8 User Interface

No source code changes are required for use of Condor, but executables must be specially linked. UNIX provides a startup routine and a library, (crt0.o and libc.a) which are automatically linked with all user programs by the UNIX compilers (cc, f77 etc.). Condor provides

its own version of crt0.o and libc.a, that have to be linked with the user program instead of the normal versions. On systems where programs are linked with shared libraries by default, the linker should be told explicitly to use static linking, since Condor does not support the use of shared libraries.
The Condor user interface consists of the programs shown in Table 2.1.

condor_submit	Submit jobs to the Condor job queue.
condor_rm	Remove jobs from the Condor job queue.
condor_prio	Change priority of jobs in the Condor job queue.
condor_q	Display the Condor job queue.
condor_globalq	Display the Condor job queue of all machines in the pool.
condor_status	Examine the status of the Condor machine pool.
condor_summary	Summarize the Condor usage on the local machine.

Table 2.1 Condor user interface.

Condor_submit reads a *description file* which contains commands that direct the queuing of jobs. It is possible to submit many Condor jobs at once, a "job cluster". These jobs must share the same executable, but may have different input and output files, and different arguments. Submitting multiple jobs in this way is advantageous, because only one copy of the checkpoint file is needed until the jobs begin execution. The description file must contain the name of the executable, and the requirements which a remote machine must meet to execute the job. Their are three kinds of requirements; "Memory", "Arch", and "OpSys". "Memory" is an estimation of the amount of physical memory that will be required, and "Arch" and "OpSys" describe the particular kind of UNIX platform for which the executable has been compiled. Another important item in the description file is the Condor job priority.

Condor jobs can be removed with the program condor_rm. Only the owner of a job or the super user can remove the job. The program condor_prio can be used to change the priorities of Condor jobs. Note that condor_prio will not preempt a job that is running, even if it's priority is lower than other jobs.

Condor_q displays information about jobs in the job queue of the machine on which the condor_q program is run. Condor_globalq displays information about all the jobs in the Condor pool. For each machine in the pool, condor_globalq displays the hostname followed by a one line summary of information for each Condor job on that machine. The summary format is as follows:

Id	The cluster/process id of the Condor job.
Owner	The owner of the job.
Submitted	The date and time the job was submitted.
CpuUsage	The accumulated remote CPU time. If the job is currently running, time accumulated during the current run is not shown.
St	Current status of the job, U=unexpended, R=Running, I=Idle (waiting for a machine to execute on), C=completed, and X=removed.

Pri	The user specified priority of the job.
Size	The virtual image size of the executable in megabytes.
ArchOS	The architecture/operating system the executable is submitted to run on.
Command	The name of the executable.

The following is an example of the output of the condor_globalq program.

```
Hostname: paramount
  ID    OWNER     SUBMITTED    CPU_USAGE  ST PRI SIZE ARCH_OS          COMMAND
337.0   andrzej   3/6  17:36   4+08:11:12 R  0   1.7  SPARC_SUNOS41    /home/p
353.0   andrzej   3/10 10:38   1+10:51:25 R  0   1.7  SPARC_SUNOS41    /home/p
364.0   ernie     3/12 10:26   0+00:02:22 C  0   0.6  SPARC_SUNOS41    /home/p
364.1   ernie     3/12 10:26   0+00:00:00 R  0   0.6  SPARC_SUNOS41    /home/p
365.0   zisis     3/12 10:58   0+00:00:00 R  0   0.6  SPARC_SUNOS41    /home/p
365.1   zisis     3/12 10:58   0+00:44:13 R  0   1.7  SPARC_SUNOS41    /home/p
365.2   zisis     3/12 10:58   0+00:58:37 I  0   1.7  SPARC_SUNOS41    /home/p
365.3   zisis     3/12 10:58   0+00:14:14 R  0   1.7  SPARC_SUNOS41    /home/p
365.4   zisis     3/12 10:58   0+00:29:24 R  0   1.7  SPARC_SUNOS41    /home/p
```

Condor_status displays information about machines in the Condor pool. For each machine, condor_status displays a one line summary with the following information:

Name	The network name of the machine.
Run	The number of Condor jobs the machine is executing remotely.
Tot	The total number of Condor jobs in the machine's job queue.
Prio	The priority of the machine for obtaining machines on which to run jobs.
State	State of the machine regarding hosting Condor jobs for others.
LdAvg	The 1 minute load average.
Idle	The keyboard idle time.
Arch	The CPU architecture of the machine.
OpSys	The operating system running on the machine.

The following is an example of the output of the condor_status program.

```
Name           Run  Tot      Prio State  LdAvg          Idle Arch    OpSys
briesje          0    0         0 Run    1.16       01:24:58 SPARC   SUNOS41
damp             0    0         0 NoJob  1.06       00:57:54 SPARC   SUNOS41
donder           0    0         0 Susp   0.38     7+21:56:26 SPARC   SUNOS41
ijs              0    0         0 NoJob  0.00       00:00:00 SPARC   SUNOS41
ijzel            0    0         0 Susp   0.79    23+21:51:15 SPARC   SUNOS41
miezer           0    0         0 NoJob  0.10       00:00:02 SPARC   SUNOS41
mist             0    0         0 Run    1.00       17:10:17 SPARC   SUNOS41
orkaan           0    0         0 Susp   0.13       00:04:33 SPARC   SUNOS41
paramount        7    8    -20323 NoJob  2.66       00:00:00 SPARC   SUNOS41
passaat          0    0         0 NoJob  1.04       00:13:58 SPARC   SUNOS41
regen            0    0         0 Run    1.00     1+22:52:20 SPARC   SUNOS41
rijp             0    0         0 Run    0.76       00:51:14 SPARC   SUNOS41
```

```
sneeuw            0    0          0 NoJob  0.00       00:02:07 SPARC   SUNOS41
stoom             0    0          0 NoJob  0.00       00:00:04 SPARC   SUNOS41
storm             0    0       -196 NoJob  0.07       00:00:00 SPARC   SUNOS41

SPARC/SUNOS41          15 machines   8 jobs   4 running
                       15 machines   8 jobs   4 running
```

Since each machine periodically sends the Central Manager a snapshot of its situation, and the information returned by condor_status is a collection of such snapshots (all taken at varying times), the total picture may not be completely consistent. For example, the reported total number of jobs running and the sum of machines reported to be serving a job should match, but they may not.

Each machine keeps information about completed Condor jobs in the history file. Condor_summary displays a summary of all the Condor jobs listed in a history file. The listing summarizes the use by each user on a single line containing the following items:

Name	The login name of the user.
Jobs	The number of jobs which have been completed for this user.
Local Cpu	The total local CPU time used to support Condor jobs for this user.
Remote Cpu	The total remote CPU time accumulated by Condor jobs for this user.
Leverage	The ratio of remote CPU time to the local CPU time.

Leverage is a new performance measure that has been introduced by Litzkow, Livny and Mutka [12]. The leverage of a job is the ratio of the capacity consumed by a job on the executing machine to the capacity consumed on the initiating machine to support remote execution. The capacity on the initiating machine is the combination of capacity used to support placement, checkpointing and system calls.
The following is a part of the output of the condor_summary program.

```
   Name   Jobs      Local Cpu     Remote Cpu       Leverage
    rvd     42     0+00:01:22     6+07:29:37         6650.9
  maart      7     0+00:12:01    24+00:00:55         2876.1
 vdhoef     27     0+00:57:54   163+00:52:24         4054.8
      .      .              .              .              .
  TOTAL    852     0+02:33:03   300+10:45:24         2826.8
```

2.9 Discussion

Three months ago the Condor pool at NIKHEF-K has been enlarged with 10 machines. The pool now consists of 18 SPARC stations. Condor has provided around 375 days of CPU time in the last 10 months. The average leverage of a Condor job in the NIKHEF-K pool is 3500, this means that only one minute of local capacity was consumed for nearly two and a half days of remote capacity. Table 2.2 shows the top ten Condor users of the NIKHEF-K pool of the last ten months.

Rank	Name	Jobs	Local Cpu	Remote Cpu	Leverage
1	vdhoef	27	0+00:57:54	163+00:52:24	4054.8
2	janv	336	0+00:15:42	64+01:28:14	5875.7
3	zisis	156	0+00:26:16	33+15:08:43	1843.7
4	maart	10	0+00:12:21	28+16:13:10	3343.6
5	ericj	36	0+00:08:34	28+11:16:50	4785.6
6	jean	21	0+00:10:36	19+02:00:26	2592.5
7	hansrp	12	0+00:08:05	12+05:40:45	2179.9
8	andrzej	17	0+00:03:18	8+04:00:20	3563.7
9	rvd	42	0+00:01:22	6+07:29:37	6650.9
10	ernst	206	0+00:12:40	3+08:58:58	383.6

Table 2.2 Top ten Condor users.

2.9.1 Security Hazards

In this section three security hazards of Condor are discussed. The first two possible ways of attack suppose that the malicious user owns a machine, that he can do anything he wants with this machine, and that he has the possibility to communicate with the machines of the Condor pool (via the Internet for example). The third possibility supposes that the malicious user has a login on a machine of the Condor pool, or that he uses one of the first two possibilities to run jobs on machines of the pool. The fact that the Condor sources are available via anonymous ftp, makes it easier for a hacker to write programs that act as Condor daemons.

- It is possible to add a machine to somebody's Condor pool if you know the network address of the Central Manager machine. This and the next security problem are caused by the fact that there is no knowledge about which machines are supposed to be part of the pool. By adding a machine to a Condor pool, a malicious user can run his jobs on machines of the pool, and he can accept Condor jobs of other users. In this way he can lay his hands on executables that may be confidential. The new machine will show up in the Condor status reports and the log files, but it can take some time before this is noticed. A malicious user can use this time to efface traces by erasing the log files or killing Condor daemons for example.

- If a Schedd of a machine, the initiating machine, has received permission from the Central Manager to run a job on another machine, the execution machine, the Schedd starts a Shadow process with as arguments the name of the machine where the job may be run and the cluster_id/job_id pair. The Shadow will contact the Startd of the execution machine with the request to run the job. The Startd has no information on whether the initiating machine has really received permission from the Central Manager, nor whether the initiating machine is part of the pool. A malicious user can start jobs on machines that run Condor, by writing a special version of the Shadow that contacts the Startd directly. The most dangerous problem is that the malicious user can decide under which UID and GID the job will run.

- Condor cannot guarantee that a remotely running user jobs won't do (unallowed) local system calls. A user is supposed to link its job with the Condor version of the start-up routine and C library (condor_rt0.o and libcondor.a). The Condor version of the C library contains (among other things) the system call stubs which do remote system calls to the initiating machine. A malicious user can write a new version of the Condor C library which does some system calls local.

The first two security hazards can be minimized by adding global knowledge to Condor about the participating machines of a Condor pool. The Condor daemons could then ignore message coming form unknown network addresses or port numbers.

The third security hazard is inherent with the concept of allowing users to run jobs on somebody elses workstation. The users of the workstations that form a Condor pool together will have to trust each other. A possible way to prevent the third security hazard is to compile and link the Condor job on the execution machine.

2.9.2 Future Work

Litzkow and Solomon [14] described the following areas of future work on Condor.

- In some circumstances it would be better to transfer processes directly between execution sites rather than always sending a checkpoint file back to the originating site.
- Data compression could be used to reduce the volume of data transferred and stored.
- The stack and data segments could be read directly from the core file into a new instantiation of a process rather than converting the core file to an executable module (which includes copying the text segment of the old checkpoint file).
- Support for signals. This is a non-trivial feature because the information maintained by the kernel has to be checkpointed. This information is maintained in a way that varies among UNIX implementations.

Unfortunately, the last two optimizations work against portability. A solution to a problem will work for some platform, but not for others.
The following is a collection of other areas of development.

- At the moment Condor is machine oriented. The central manager is responsible for prioritizing the machines of a pool, and the Schedd of each machine has as task to prioritize the jobs submitted on that machine. This mechanism only works well if each user would always submit his/her jobs from the same machine. In practice this is not always the case due to the possibility of remote login. Another problem is multi-user systems that are shared by a large group of people. These people now have a shared priority that will be very low. In the future, Condor will become user oriented. There are two possible architectures. One possibility is to give each user his own queue of Condor jobs. In this situation the same scheduling mechanism can be used, each user has a priority and it is the responsibility of the user to prioritize its jobs. The second

possibility is to make one global queue for all the jobs of all users. This means that a new scheduling mechanism for the jobs in the global queue will be needed.

- Support for shell scripts. The possibility to run a shell script is interesting, because in this way one can have the script compile and link the job on the fly on the remote machine. This is useful in a multi-architecture environment. It should be possible to let the script end with the execution of a checkpointed program.

- The possibility to submit a number of programs connected by pipes as one Condor job. A test version of this feature has recently been implemented in Wisconsin on request of the CERN/SMC project. They have simulation programs and data processing programs that simulate each a part of an experiment. These programs can be configured together in a pipeline for simulating a whole experiment. The problem with submitting the programs as separate Condor jobs is that a program cannot be executed before the previous program in the pipeline has generated its output. This problem is solved when the programs are submitted as one Condor job.

Chapter 3

The Condor Control Software

This chapter describes some of the data structures and algorithms used by the Condor control software. In the first section we look at the most important data structure of Condor, context. This data structure is used to describe characteristics of jobs and machines, and to store the control functions from the Condor configuration files. Section 2, 3, and 4 describe the data structures of the Startd, Schedd, and Collector, respectively. Section 5 contains 'pseudo' code which indicates how the Negotiator does the scheduling. Finally, Section 6 describes exactly how a Condor job is started.

3.1 The Context Data Structure

Condor can be customized by means of the Condor configuration files (See Section 2.5). Every daemon will, as part of its initialization process, configure itself by reading in the condor configuration files. The macros are inserted into a configuration table, and a *context* is created from the control functions. The daemon uses the configuration table to initialize parameters that are relevant for it. Context is a list of expressions, an expression is a list of elements, and an element can be a string, an integer or a floating-point number. Context is defined as:

```
typedef struct {
        int             len;
        int             max_len;
        EXPR            **data;
} CONTEXT;

typedef struct {
        int             len;
        int             max_len;
        ELEM            **data;
} EXPR;
```

```
typedef struct {
        int             type;
        union {
                int     integer_val;
                float   float_val;
                char    *string_val;
        } val;
} ELEM;
```

An example of the context created when a daemon starts up is:

```
OpSys       = SUNOS41;
Arch        = SPARC;
Machine     = hoos;
START       = LoadAvg <= 0.300000 && KeyboardIdle > 15 * 60;
SUSPEND     = LoadAvg >= 1.500000 || KeyboardIdle < 5;
CONTINUE    = LoadAvg <= 0.300000 && KeyboardIdle > 5 * 60;
VACATE      = CurrentTime - EnteredCurrentState > 10 * 60;
KILL        = CurrentTime - EnteredCurrentState > 10 * 60;
PRIO        = UserPrio * 10 + Status != 0 - QDate / 1000000000.000000;
INACTIVE    = Users <= 0;
UPDATE_PRIO = Prio + Users - Running;
```

This context will be called the *start-up* context of a daemon. The first three expressions describe the machine on which the daemon is running. The expressions START, SUSPEND, CONTINUE, VACATE, and KILL are used by the Startd and define when it should change state. The Schedd uses the expression PRIO to calculate the priority of user jobs, and the expressions INACTIVE and UPDATE_PRIO are used by the Negotiator to update the priority of the machines. The data structure context is also used by the Schedd and Startd to send information about the state of a machine to the Collector and additionally by the Schedd to send information about the jobs to the Negotiator.

3.2 The Start-Daemon

The Startd evaluates the load on the local machine every five seconds, and makes decisions whether to suspend, resume, or abort a foreign job, if one is running on the machine. Every two minutes the Startd will update the Central Manager regarding conditions on the local machine. It sends a message to the Collector consisting of the start-up context plus the following expressions:

State	State of the Startd (NoJob, Running, Suspended, Vacating, or Killed).
EnteredCurrentState	Time when current state was entered (seconds since 1-1-1970).
Memory	Physical memory installed on the machine in megabytes.

Disk	Free diskspace in kilobytes on the filesystem where foreign checkpoints are stored.
KeyboardIdle	Time in seconds since last activity on any tty or pty.
Cpus	Number of CPUs installed.
LoadAvg	Current value of the UNIX one-minute load average.
VirtualMemory	Swap space available on the machine in kilobytes.
Subnet	Name of the subnet the machine is on in "dot notation".
MaxJobsRunning	Maximum number of jobs this machine will run at any one time.

3.3 The Sched-Daemon

The Schedd is responsible for maintaining the queue of Condor jobs that have been submitted from the machine where the Schedd is running. The queue consists of clusters of related processes. All processes in a cluster share the same executable (and thus the same initial checkpoint), and are submitted as a group via a single description file. Individual processes are identified by a cluster_id/proc_id pair. Cluster_ids are unique for all time, and process_ids are in the range 0 - n when there are n processes in the cluster. Individual processes are not removed from a cluster, they are just marked as "completed", "deleted", "running", "idle", etc. A Cluster is removed when all processes in the cluster are completed, but the cluster_ids are not re-used.

The low-level database operations are accomplished by the ndbm(3) library. The functions in this library maintain key/content pairs in a database. The cluster_id/proc_id pair is used as key. The database is stored in two files. One file is a directory containing a bitmap and has **.dir** as its suffix. The second file contains data and has **.pag** as its suffix. By storing the job queue on disk, no information will be lost if a Schedd dies. The queue consists of job records that are defined as follows:

```
typedef struct {
     int             version_num;     /* version of this structure */
     PROC_ID         id;              /* job id (cluster and proc) */
     char            *owner;          /* login of person submitting job */
     int             q_date;          /* UNIX time job was submitted */
     int             completion_date; /* UNIX time job was completed */
     int             status;          /* Running, unexpanded, completed, .. */
     int             prio;            /* Job priority */
     int             notification;    /* Notification options */
     int             image_size;      /* Size of the virtual image in K */
     char            *cmd;            /* a.out file */
     char            *args;           /* command line args */
     char            *env;            /* environment */
     char            *in;             /* file for stdin */
     char            *out;            /* file for stdout */
     char            *err;            /* file for stderr */
     char            *rootdir;        /* Root directory for chroot() */
```

```
        char            *iwd;           /* Initial working directory   */
        char            *requirements;  /* job requirements */
        char            *preferences;   /* job preferences */
        struct rusage   local_usage;    /* accumulated usage by shadows */
        struct rusage   remote_usage;   /* accumulated usage on remote hosts */
} PROC;
```

The Condor jobs are placed in the job queue by the command *condor_submit* and can be removed from the queue by the command *condor_rm*. It is possible to change the priority of a job in queue with the command *condor_prio*. With the command *condor_q* it is possible to get an overview of only the local queue, with the command *condor_globalq* one gets an overview of all the queues. Every five minutes the Schedd checks the local job queue and updates the Central Manager by sending a context to the Collector consisting of the start-up context and the following expressions:

Running	Number of jobs.
Idle	Number of idle jobs in the queue.
Users	Number of different users with jobs in the queue.

3.4 The Collector

The Collector manages three data structures:

- a priority database,
- a linked list of machine records,
- an array of status lines.

The priority database is implemented with the ndbm(3) library. The network address of a machine is used as key. The database is read in when the Collector starts up, and saved every time the Negotiator has updated the priorities. In this way the priorities will not be lost if the Collector dies. Periodically, the Schedd and the Startd send a message to the Collector. With this information the Collector updates two data structures. The first of these is a linked list of records, each containing information about a machine. Such a machine record is defined as:

```
typedef struct mach_rec {
        struct mach_rec *next;
        struct mach_rec *prev;
        char            *name;
        struct in_addr  net_addr;
        short           net_addr_type;
        CONTEXT         *machine_context;
        int             time_stamp;
        int             prio;
        int             busy;
        STATUS_LINE     *line;
} MACH_REC;
```

The members name, net_addr and net_addr_type are set when the Collector receives information for the first time about a machine. Time_stamp is the most recent time at which the Collector received information from the Schedd or Startd of a machine. It is used to determine if Condor is still running on that particular machine. The prio member contains the priority of the machine and is consistent with the value in the priority database. The busy member is used by the Negotiator when it is scheduling, to indicate that a Condor job has already been scheduled to this machine. The member machine_context is a combination of the context received from the Schedd and the Startd. The member STATUS_LINE contains a selection of the information from the machine_context and is defined as:

```
typedef struct status_line {
        char            *name;
        int             run;
        int             tot;
        int             prio;
        char            *state;
        float           load_avg;
        int             kbd_idle;
        char            *arch;
        char            *op_sys;
} STATUS_LINE;
```

The second data structure that the Collector updates, is an array with status lines of all machines. This array is used when a user asks the status of the Condor pool with the command *condor_status*.

3.5 The Negotiator

Every 5 minutes the Negotiator allocates the idle machines to machines which have Condor jobs to run. The following 'pseudo code' indicates how the Negotiator does the scheduling. First of all the Negotiator asks all machine records from the Collector. The Negotiator updates the priorities and sorts the machine records in order of priority. The Negotiator first schedules the machine with the highest priority.

```
 RESCHEDULE
 1) send Collector GIVE_STATUS command
 2) receive all machine records
 3) update priorities
 4) sort machine records in order of priority
 5) for (machine with highest priority to machine with lowest priority)
 6)    if (machine information is up to date and the machine has idle jobs)
 7)       send Schedd NEGOTIATE command
 8)       while (priority is higher then priority of the next machine and
                 this machine has idle jobs)
 9)          SIMPLE NEGOTIATE
10)       send Schedd END_NEGOTIATE command
11) send Collector NEGOTIATOR_INFO command
12) send priorities to the Collector
```

The Negotiator asks the Schedd of this machine, by means of the SEND_JOB_INFO command, to send information about the job it wants to run. The Schedd will respond by sending information about the job with the highest priority, or the command NO_MORE_JOBS. The job information consists of the job requirements, the job preferences, and the login-name of the owner of the job. The job requirements is an expression which states the architecture and operating system needed, the physical memory the server should have, and an estimation of the required disk-space and virtual memory.

```
SIMPLE NEGOTIATE
 1) while (not SUCCUSS and not FAIL)
 2)    send Schedd SEND_JOB_INFO command
 3)    receive operand
 4)    case JOB_INFO
 5)       receive job_context
 6)       FIND SERVER
 7)       if (FOUND SERVER)
 8)          send Schedd PERMISSION command and name of server
 9)          decrement the counter of idle jobs for this machine
10)          decrement the priority of this machine
11)          mark server as busy
12)          return SUCCESS
13)       else
14)          send Schedd REJECTED command
15)    case NO_MORE_JOBS
16)       return FAIL
```

The Negotiator will try to find a server for the job by walking through the list of machines in search of a machine that is idle, not running a Condor job, and that both meets the job requirements and job preferences. If no server was found, the Negotiator walks a second time through the list of machines, this time without trying to meet the job preferences.

```
FIND SERVER
 1) for (first machine in MachineList to the last one)
 2)    if (the machine is not running a Condor job, and the machine is idle,
           and the machine meets the job requirements and job preferences)
 3)       return FOUND SERVER
 4) for (first machine in MachineList to the last one)
 5)    if (the machine is not running a Condor job, and the machine is idle,
           and the machine meets the job requirements)
 6)       return FOUND SERVER
 7) return NOT_FOUND_SERVER
```

If the Negotiator has found a server for the job, it sends the name of the server to the Schedd, otherwise the command REJECTED. The Negotiator decrements the priority of the machine that it is negotiating with by one if a server was found. The Negotiator tries to find servers for the jobs of the machine it is negotiating with until the priority of the machine is lower than the following machine (in order of priority), or all the jobs in the queue of the machine have been tried. When the Negotiator has negotiated with all machines that wanted to run jobs, it sends the updated priorities to the Collector.

3.6 The Starter and Shadow Processes

The following list describes what happens when a Condor job is started.

1. The Schedd has received the PERMISSION command and the name of the server from the Negotiator. It starts a Shadow process with as arguments the name of the server and the cluster_id/process_id pair. The Schedd marks the job as running (in the queue) and keeps a 'Shadow record' which consists of the PID of the Shadow, the cluster_id/process_id pair and the name of the server.

2. The Shadow reads the job record from the job queue and sends the START_FRGN_JOB command and a context consisting of the job requirements, job preferences and the owner of the job to the Startd of the server.

3. The Startd of the server receives the context and checks if the machine and job meet each other's requirements. If so, it sends the OK command to the Shadow and changes its state to JOB_RUNNING, otherwise it sends the NOT_OK command to the Shadow.

4. The Startd creates two stream socket ports, sends the port numbers to the Shadow process and waits until the Shadow has connected to the ports. It then starts the Starter process with as argument the name of the initiating machine.

5. The Shadow sends the job record to the Starter (via the ports created by the Startd). The Shadow now executes a fork. The parent runs under the UID of Condor and takes care of updating the information in the job queue, sending mail upon completion of the job, etc. The child handles the remote system calls and runs with the UID of the owner of the job.

6. The Starter copies the checkpoint file from the initiating machine to the execution machine, either by NFS or via the Shadow with a so called 'pseudo' remote system call. Then the Starter executes a fork. The parent sets the alarm for the current checkpoint interval and executes a wait3 system call. The child sets its UID and GID to those of the owner of the job and executes an exec system call to start the user job. The parent is responsible for restarting the job after a checkpoint, sending remote usage information to the shadow, sending back the checkpoint file when the job was aborted, etc.

Figure 3.1 shows the situation when the Condor job is running.
The Startd and the Starter only communicate via signals. The Startd can send the Starter a signal to checkpoint, suspend, resume, vacate or kill the job. When the Starter exits, the Startd will change its state to NO_JOB. The Schedd and the Shadow communicate only indirectly through the job queue. The Shadow updates the information in the job queue, and marks the job as IDLE if the Condor job had to vacate the remote machine, REMOVED if the owner removed the job from the queue, and COMPLETED if the job has completed.

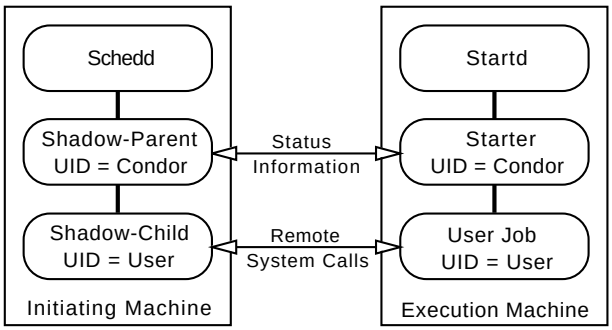


Figure 3.1 Starter and Shadow processes.

Chapter 4

Condor Flocking

A *Condor flock* is a collection of Condor pools that share the load of Condor jobs in some way, to provide a better turn-around time to some or all users. The terminology used in this chapter was invented by Miron Livny.

The first situation at which we will look in this chapter is when a user has the right to submit his Condor jobs in more than one pool. This user will want to distribute his jobs in such a way over the pools that the total turn-around time is minimal. This kind of flock is called a *private flock* because it is a flock for one user.

The second situation is when two or more owners of Condor pools (institutes working together on a project, departments of a company, etc.) have decided to couple their pools so that jobs of one pool can be executed in another pool. This will be called a *group flock* because it is a flock for all the users of a pool. We call the pool where the job was submitted the *initiating pool*, and the pool where the execution machine is situated the *execution pool*.

In both of these situations it is possible to make a distinction between an *administrative relation* and a *physical relation* between pools. The distinction is based on the delay of inter-process communication between machines of two different pools. If the delay is low enough to allow remote system calls (comparable to the delay of inter-process communication between machines of one pool), we talk about an administrative relation. If the delay is high, the remote system call mechanism will no longer be usable in most cases, and another mechanism should be used to preserve the local execution environment for a remotely executing process. In this case we talk about a physical relation.

4.1 A Private Flock

When a user has the possibility to submit Condor jobs in several pools, he or she will have to choose in which pool to submit his jobs. If the user submits his jobs in only one pool, it is possible that Condor jobs of this user are waiting, while in another pool they could be run. The jobs of this user may be waiting for an idle machine, or they may be waiting because there are other Condor users with a higher priority, that want to run jobs. Another problem of submitting the jobs in only one pool is that the priority of this user will only increase in

the pool where he has submitted the job. It is possible to submit the jobs in all the pools, but it is undesirable that duplicates of a job are running. These duplicates will waste resources and they will decrease the priority of the user which means that his other jobs will have to wait longer. Another problem is that the user will have to make duplicates of the input and output files.
A possible way to implement a private flock is:

- Jobs are submitted in all the pools, so that the priority of the user will be increased in all the pools as long as a job is not running.
- When a job is scheduled in one of the pools, the job should be marked in the other pools so that it will not be run, and no longer influences the priority of the user in these pools.
- When a job finishes, the job is removed from all the queues.

4.2 A Group Flock

Within a Condor pool there is a wide variation in the number of Condor jobs that are waiting for service. Sometimes there is a large collection of jobs, and sometimes there are no jobs while there are idle machines. By coupling Condor pools, work from "overloaded" pools can be transferred to pools with idle machines. This means that agreement should be reached between the owners of Condor pools. There are several kinds of agreements that can be made between the owners of Condor pools:

- When there are idle machines in a pool that cannot be used for jobs of users of that pool, then these machines may be used for jobs of users of other pools. This kind of agreement is similar to the agreement between owners of workstations within a Condor pool. A workstation may only be used when the owner does not need it.
- If pool A uses some machines of pool B now, it gives pool B the right to use some machines of pool A in the future. This may mean that a Condor pool has to run jobs from another pool to pay back for used capacity in the past, even when there are jobs that were submitted in this pool.

The initiating pool should decide when and where to move a Condor job. The execution pool has to decide which machines may be used and with which priority the "foreign" Condor job should run.

4.3 General Design of the Group Flock

This section describes the general design of the Condor group flock. This is the ideal design of the group flock as we see it. Chapter 5 describes what has been implemented during my six month graduating term.

4.3.1 Centralized versus Distributed Approach

The first design issue involves whether one machine should collect the status information of all the Condor pools and take the decisions, or whether the decision-making process is to be distributed among the Condor pools. The most important feature of making decisions centrally is simplicity. However, an important drawback of the centralized approach is the low reliability of such system; failure of the machine which takes the decisions results in the collapse of the entire management system.

Therefore we decided that in the Condor group flock situation, it should still be the Central Manager of a pool who allocates the idle machines to waiting Condor jobs. In the normal Condor pool situation the Central Manager allocates idle machines of the pool to waiting Condor jobs. In the Condor group flock situation the set of idle machines is enlarged with the set of idle machines of other pools that may be used. The set of waiting jobs is enlarged with waiting Condor jobs from the other pools of the flock. Figure 4.1 shows the new functionality of the Central Manager.

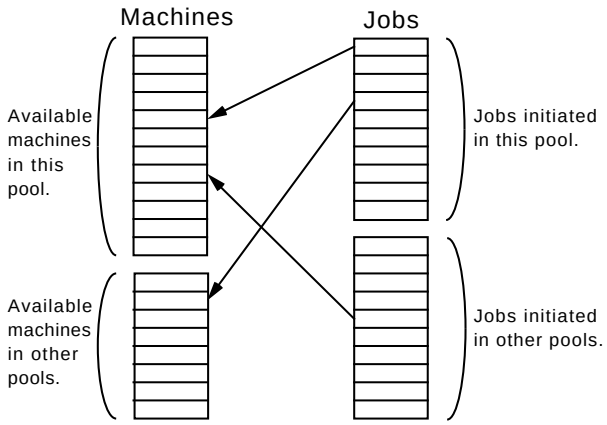


Figure 4.1 New Central Manager functionality.

Each Condor pool may have different policies that determine which idle machines may be used by which Condor pool. The Central Manager may have different policies for prioritizing the waiting Condor jobs, and different scheduling algorithms that determine which Condor job is executed on which machine.

4.3.2 Source-initiative versus Server-initiative Approach

There are two possible ways in which Condor pools can decide to move a Condor job from one pool to another pool, a source-initiative and a server-initiative way.

In the source-initiative approach, a Condor pool goes searching for a server when it has a Condor job which it wants to run on a machine of another pool. The initiating pool may decide that a job should be run on a machine of another pool, when there are no idle machines in the own pool that can serve this job, or when it expects that there is a better (faster) server available in one of the other pools. The following is an example of a source-initiative algorithm.

1. The initiating pool sends a request for an idle machine that can serve the job, to the other Condor pools of the flock.
2. When an execution pool receives this request and decides that one of its machines may be used for this job, it tells so with a message to the initiating pool.
3. The initiating pool can then decide whether it will run the job on this machine.

In the server-initiative approach, the Condor pool that has idle machines that may be used for jobs of the other Condor pools of the flock, goes searching for work. The following is an example of a server-initiative algorithm.

1. A pool that has idle machines that may be used by the other Condor pools, sends a message to these pools, advertising the idle machines.
2. When the initiating pool schedules the waiting Condor jobs, it may decide that a job can be best served by one of the advertised machines. The initiating pool will ask permission from the execution pool to use one of its idle machines for the Condor job.
3. The execution pool can then decide whether it will let the initiating pool use the idle machine.

We have chosen the server-initiative approach, because in this way the Central Manager of a pool knows which machines are available in the flock when it starts scheduling. The Central Manager can therefore schedule the waiting Condor job with the highest priority to the best server available in the flock.

4.3.3 Server-values

A powerful idea is to give every machine a server-value. This value should give an expression of the relative processing power of the server. For every platform there should be a standard assignment of values, which is used by all the pools that are part of a flock, so that servers of different pools can be compared. Server values make it possible for the Central Manager to compare all the idle machines of the flock and to choose the best server for a job. It is possible to take all kind of factors into account, for example the time that the server is already idle and the average length of an idle period of this machine. It is also possible to adjust the server-values of the idle machines of the other pools for the expected performance loss due to transmission delays of remote system calls, higher cost of migration, etc.

4.3.4 User Identity

Within a Condor pool, jobs run under the UID and GID of the owner of the job, so that files can be accessed via NFS. In the Condor flock situation it will normally not be possible to run the jobs under the UID and GID of the owner, because the owner has no UID on the machines of the execution pool. In the flock situation, jobs can be run under the UID "nobody" (this was the case before the NFS optimization) or under one or several special Condor flock UID's.

In this case, files that could be accesses via NFS from the execution machine, now have to be accessed via remote system calls because the job doesn't run under the UID of the owner.

Chapter 5

Design of a Group Flock

My assignment consisted of designing and implementing a first version of an administrative group flock, that is, a group flock where the relations between all the pools are administrative. In an administrative group flock it is possible to use the remote system call mechanism to preserve the local execution environment for remotely executing jobs. Neither are changes required to the checkpointing mechanism. The Condor control software should be changed or extended in such a way that it is possible that jobs of one Condor pool are served by idle machines of other Condor pools.
The following requirements were posed to the design:

- It should be transparent for the user (except for somewhat poorer performance) whether a job is executed in the initiating pool or in another Condor pool.
- It should be possible for a user to prevent that one of his Condor jobs is executed in another pool.
- A Condor pool has a set of pools that are allowed to run jobs on the machines of the pool, and a set of pools where this particular pool is allowed to run jobs. It should be possible to change the relation between pools.
- Failure of processes concerned with Condor flocking should not affect the working of the Condor daemons. In other words, failure of Condor flocking should not make it impossible for the Central Manager to allocate idle machines of the pool to Condor jobs of the initiating pool.

An additional very important restriction is that nothing may be changed to the Central Manager, and as little as possible to the other Condor software. This restriction was posed by Miron Livny to prevent the existence of several versions of the Central Manager, which would make providing support for the Condor design team very difficult. The design presented in this chapter is therefore a compromise between the ideal Condor group flock situation as presented in Chapter 4, and the restriction to change as little as possible to the existing Condor control software. Only the Schedd has been changed.

Section 1 of this chapter gives an overview of the design. Section 2 describes the limitations of the design, and the assumptions that were made. The other sections describe particular parts of the design.

5.1 The World Machine

The basic idea is to implement the group flock by adding a virtual machine to each Condor pool, called the *World Machine*. This virtual machine represents in the local pool all the other Condor pools that are part of the flock. The World Machine looks to the Central Manager like a normal machine in the way that it can offer jobs for execution, and can serve jobs of machines of the pool. The World Machine consists of two daemons, the *W-Schedd* and the *W-Startd*. Figure 5.1 shows three Condor pools, which together form a flock.

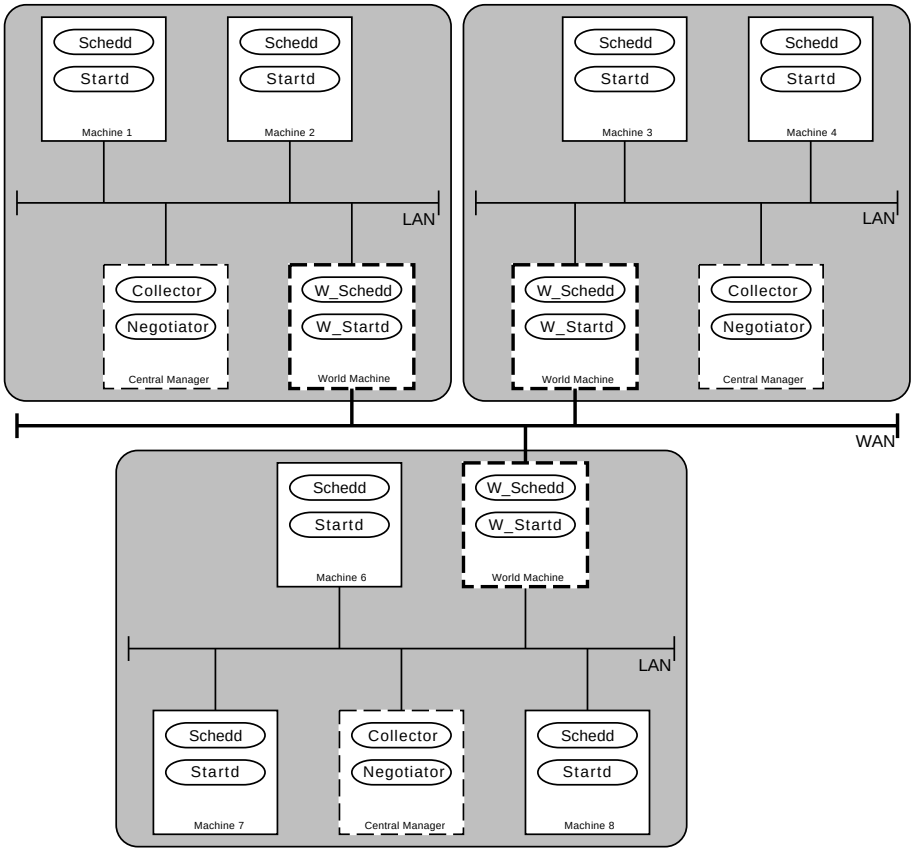


Figure 5.1 A Condor flock.

We want the Central Manager of the initiating pool to give permission to machines of that Condor pool to run jobs on the World Machine, and the Central Manager of the execution pool to give permission to its World Machine to run jobs on machines of the execution pool. To achieve that the Central Manager gives permission to machines of its pool to run jobs on

the World Machine, the W-Startd presents itself to the Central Manager as an idle machine. Periodically, the W-Startd daemons of a flock exchange information about the idle machines in their pool. The W-Startd chooses one of the idle machines of the other pools, and presents itself to the Central Manager as a machine with the same characteristics as this idle machine.

When the Central Manager has given permission to one of the machines of its pool to run a job on the World Machine, the Schedd that runs on this machine will ask the W-Startd to give it the name of a real server on which it may start the job. The W-Startd will then request a idle machine from the W-Schedd of a pool that has idle machines that can serve this job.

So the W-Schedd receives requests for idle machines from the other World Machines. To achieve that the Central Manager of the execution pool gives permission to the World Machine to use machines of the pool, the W-Schedd acts like a normal Schedd that has jobs in the queue. The W-Schedd periodically tells the Central Manager how many jobs it wants to run, and negotiates with the Central Manager in the same way as a normal Schedd. If the W-Schedd receives permission from the Central Manager to use a machine, it tells so to the W-Startd of the initiating pool, who tells it to the Schedd. The Schedd will start a Shadow process that starts the job on the machine of the execution pool, in the same way as when this machine had been part of the initiating pool.

5.2 Limitations and Assumptions of this Design

The restriction that nothing may be changed to the Central Manager implies the following limitations.

- The W-Schedd and the W-Startd have to use the same port numbers as the normal Schedd and Startd processes, therefore it is not possible to run the daemons of the World Machine on a machine that is part of the pool. A machine should be given the role of World Machine.
- The priority of the World Machine will be calculated by the Central Manager in the same way as the priority of a normal machine.
- The Negotiator will give permission to only one machine per schedule interval, to run a job on the World Machine.

The assumption has been made that the pools (of a flock) consist of machines that have the same architecture and operating system. This assumption is necessary because the World Machine presents itself to the Central Manager as one of the idle machines of the other pools. If there would be different machine types in the flock then the probability that there is a job that can run on the World Machine would be very small.

The question under which UID the Condor jobs from another pool should run has been avoided by demanding that the owners of these jobs have an account on the machines of the execution pool. This means that the Condor jobs can run with the UID of the owner.

The remainder of this chapter explains how the World Machines and the Condor daemons work together to start jobs in other pools. The implementation of the W-Startd and the W-Schedd is described in the next chapter.

5.3 Flock Configuration

Information about the pool configuration is stored in a special file, the *flock configuration file*. This file contains for every pool of the flock the following information:

- The name of the pool.
- The network address of the World Machine.
- Whether or not that pool is allowed to run Condor jobs in this pool.
- Whether or not this pool is allowed to run Condor jobs in that pool.

The following parameters are defined in the Condor configuration files:

- The name of the machine where the W-Startd and the W-Schedd run (the World Machine).
- The name of the flock configuration file.
- The name and the length of the logfiles. The W-Startd and the W-Schedd log important actions and write debug information in these logfiles.
- The value of the debug flags. The value of these debug flags decides what kind of information is written into the logfiles.
- The interval at which the W-Startd determines the state of the Condor pool, sends information about idle machines to the other pools, and updates the Central Manager.
- The interval at which the W-Schedd updates the Central Manager.

5.4 Information Exchange

The W-Startd periodically determines the state of the pool. It sends the command GIVE_STATUS to the Collector, which replies by sending all the machine records.

With this information the W-Startd makes a list of the idle machines in the pool. The information about the idle machines consists of the name, address, and the context of the machine. A machine is considered to be idle if the START expression is true, and the state of the machine is NOJOB. The W-Startd sends the list of idle machines to all the other W-Startd processes of pools that have the right to run jobs in this pool.

The W-Startd receives lists of idle machines from the other W-Startd processes. The W-Startd will keep the latest received list of idle machines from each pool in a list. If the

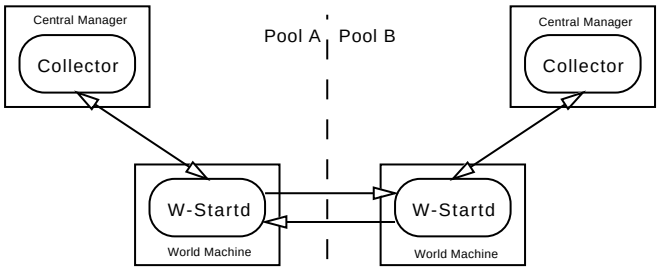


Figure 5.2 Information exchange.

W-Startd has not received a new list of idle machines from a pool that is part of the flock during a certain period, then this pool is considered to be "down". The list of idle machines of this pool will then be deleted.

5.5 Starting a Job in Another Pool

Figure 5.3 shows the situation where a machine receives permission from the Central Manager to run a job on the World Machine.

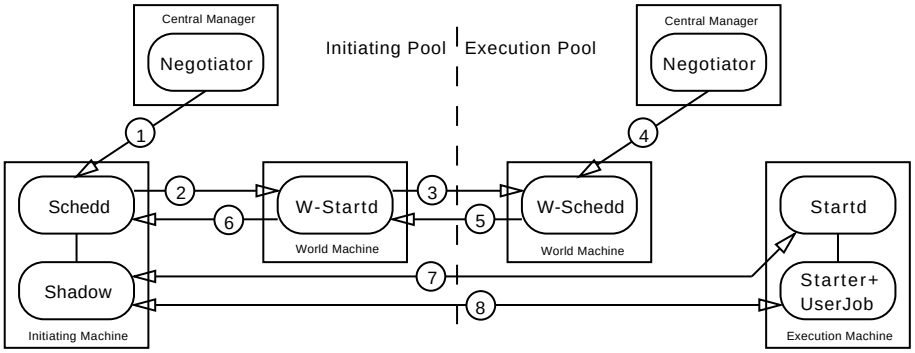


Figure 5.3 Flocking a job.

The following actions are taken by the daemons to start the job in another pool.

1. The Schedd of the initiating machine receives permission (during the negotiations) from the Central Manager to run a job on the machine "world", the World Machine.

2. The Schedd marks the job as running (in the queue). The Schedd reads the job record from the job queue and sends the GIVE_MACHINE command and a context consisting of the job requirements, job preferences and the owner of the job to the W-Startd. The Schedd does not wait for the reply, because it is negotiating with the Negotiator.

3. The W-Startd receives the job context and checks if there is a idle machine in another pool that meets both the job requirements and job preferences. If no machine was found, the W-Startd searches for a idle machine that only meets the job requirements.

If still no machine was found, the W-Startd sends the command NO_MACHINE to the Schedd, which marks the job as idle. If a idle machine was found, the W-Startd sends the command REQUEST_MACHINE and the job context to the W-Schedd of the pool with the idle machine.

4. The W-Schedd tells its Collector that it wants to run a job and waits for the Negotiator to schedule. The W-Schedd negotiates with the Central Manager in the same way as a normal Schedd.

5. If the W-Schedd receives the PERMISSION command, it sends the FOUND_MACHINE command and the name of the execution machine to the W-Startd that requested the server, if the W-Schedd receives the REJECTED command it will send the NOT_FOUND command to the W-Startd.

6. If a server was found, the W-Startd sends MACHINE and the name of the execution machine to the Schedd of the initiating machine, otherwise it sends the NO_MACHINE command to the Schedd.

7. If the Schedd receives the NO_MACHINE command, it marks the job as idle so that it can be scheduled the next time the Central Manager schedules. If the Schedd receives the MACHINE command, it starts a Shadow process for the job, with as arguments the cluster_id/process_id pair of the job, and the name of the execution machine. The Shadow will send the START_FRGN_JOB command and a context consisting of the job requirements, job preferences and the owner of the job to the Startd of the execution machine. If the situation on the execution machine hasn't changed since the last update of this machine to its Central Manager, the Startd will respond with an OK.

8. The job is now started by the Shadow and the Startd in the same way as described in Section 3.6.

All the messages concerned with starting a job contain the name of the initiating machine and the cluster_id/process_id pair of the job. This is necessary because the daemons of the World Machines may be working for many jobs of different machines and different pools at the same time.

Chapter 6

The Implementation of the World Machine

This chapter describes the data structures of the W-Startd and the W-Schedd, and the actions they perform upon reception of the different commands. The W-Startd and W-Schedd have been implemented as servers, which are activated when a process connects to their socket, or when a certain period has elapsed. All messages begin with an integer called the command, which indicates the required action. The communication between the Condor daemons is via stream sockets, except for the periodic SCHEDD_INFO and STARTD_INFO messages from the (W-)Startd and (W-)Schedd to the Collector, which are via datagrams. All communication is done with use of the eXternal Data Representation (XDR) to ensure that data is represented the same way on the different platforms.

6.1 External Data Representation

XDR is a set of library routines that allow a C programmer to describe arbitrary data structures in a machine-independent fashion. XDR library routines should be used to transmit data that is accessed (read or written) by more than one type of machine. A program can use XDR to create portable data by translating its local representation to the XDR standard representation; similarly, a program can read portable data by translating the XDR standard representation to its local equivalents.

A record stream is an XDR stream built on top of the UNIX file or 4.2 BSD connection interface. The routine **xdrrec_create(xdrs, sendsize, recvsize, iohandle, readproc, writeproc)** provides an XDR stream interface that allows for a bidirectional, arbitrarily long sequence of records. The routine initializes an XDR stream pointed to by **xdrs**. The steam does its own data buffering. The parameters **sendsize** and **recvsize** determine the size in bytes of the output and input buffers. When a buffer needs to be filled or flushed, the routine **readproc()** or **writeproc()** is called, respectively, with as parameters the iohandle, a pointer to the buffer, and the number of bytes that need to be transferred. The programmer has to write these routines. The XDR stream is destroyed by the routine **xdr_destroy(xdrs)**.

The contents of the records are meant to be data in XDR form. The XDR library provides primitives to translate between numbers, C's floating point types, strings, arrays, unions, etc, and their corresponding external representations. For each data type, **xxx**, there is an associated XDR routine of the form **xdr_xxx(xdrs, xp)**, where **xp** is a pointer to **xxx**. The routines return FALSE if it fails, and TRUE if it succeeds. XDR routines are direction independent, that is, the same routines can be called to encode or decode data.

The XDR stream provides means for delimiting records in the byte stream. The routine **xdrrec_endofrecord()** causes the current outgoing data to be marked as a record. The routine **xdrrec_skiprecord()** causes an input stream's position to be moved past the current record boundary and onto the beginning of the next record in the stream. The routine **xdrrec_eof()** returns TRUE if there is no more data in the stream's input buffer.

6.2 Signal Handlers

The W-Startd and the W-Schedd both have signal handlers for the ALARM, SIGPIPE, SIGINT and SIGHUP signal. These signal handlers have the following tasks.

ALARM The W-Startd and the W-Schedd set the alarm when a daemon connects to their socket (the client socket). The alarm is set to prevent that the W-Startd or W-Schedd will wait forever when the daemon that connected to the socket is supposed to send some information, but doesn't do so. The W-Startd also sets an alarm when it requests the status records from the Collector. The alarm is stopped if the connection is closed. When the alarm goes off, the connection is closed so that the **read()** will fail. The service routine involved will clean up.

SIGPIPE If the W-Startd or W-Schedd is communicating with another daemon and the connection goes away, it receives a SIGPIPE signal. The signal handler will close the corresponding connection, so that the **write()** will fail. The service routine involved will clean up.

SIGHUP The SIGHUP signal can be used to make the W-Startd or the W-Schedd re-read the Condor configuration files and the flock configuration file. In this way it is possible to change the relations between pools of the flock, without having to restart the daemons of the World Machine.

SIGINT The W-Startd and the W-Schedd write a last message in the logfiles and exit when they receive the SIGINT signal.

6.3 Flock Configuration

When the W-Startd and the W-Schedd start up, they read in the Condor configuration files and the flock configuration file. The information from the flock configuration file is stored in an array that consists of records defined as follows.

```
typedef struct config_rec {
        char            *pool_name;
        char            in;
        char            out;
        struct in_addr  net_addr;
        short           net_addr_type;
} CONFIG_REC;
```

The member pool_name contains the name of the pool. The members in and out contain the value TRUE or FALSE, indicating whether the pool may run jobs in our pool, and whether the pool will accept our jobs respectively. The members net_addr and net_addr_type contain the address of the World Machine of the pool.

6.4 The W-Startd

The W-Startd manages one big data structure that is used to store information about idle machines in the other pools of the flock. When the W-Startd starts up it creates a FREE_MACH_LIST record for every pool where this pool has the right to run jobs.

```
typedef struct free_mach_list {
        struct free_mach_list *next;
        struct free_mach_list *prev;
        char            *pool_name;
        int             time_stamp;
        FREE_MACH       *free_machines;
} FREE_MACH_LIST;
```

The member time_stamp is used to save the most recent time at which the W-Startd received a list of idle machines from this pool. The member free_machines is a pointer to a linked list of free-machine records. The records of this list are defined as follows.

```
typedef struct free_mach {
        struct free_mach *next;
        struct free_mach *prev;
        char            *name;
        CONTEXT         *machine_context;
} FREE_MACH;
```

The members name and machine_context contain the name and context of the idle machine. The W-Startd stores information about the requests that it is serving in SERVING records. These records are defined as follows.

```
typedef struct serving {
        struct serving  *next;
        struct serving  *prev;
        char            *machine_name;
        PROC_ID         id;
        char            *pool_name;
} SERVING;
```

The member machine_name contains the name of the machine that requested a server and the member id contains the cluster_id/job_id pair of the job for which the server was requested. The member pool_name contains the name of the pool to which the W-Startd has sent the REQUEST_MACHINE message for this request.

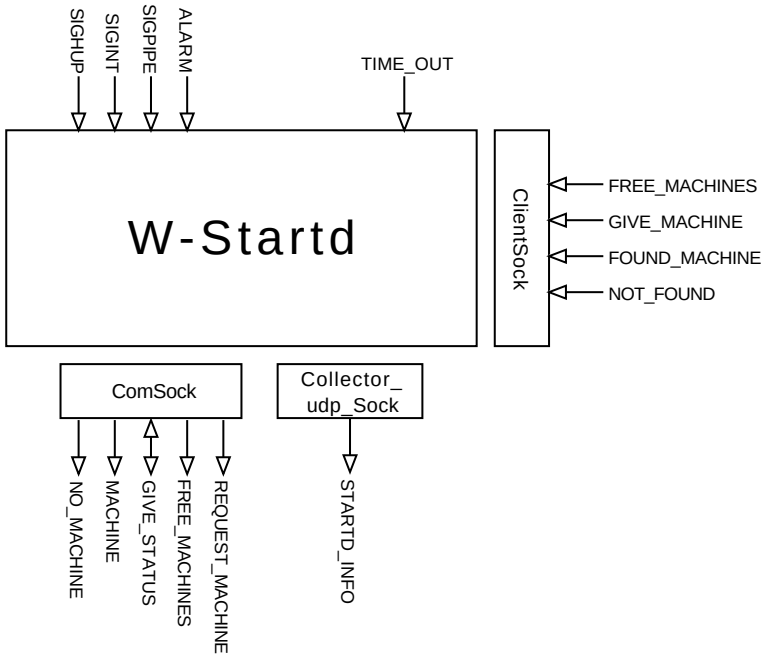


Figure 6.1 W-Startd activation.

Figure 6.1 shows that the W-Startd is activated when a certain period of time has passed, or one of the following messages is received.

- A FREE_MACHINES message from another W-Startd.
- A GIVE_MACHINE message from a Schedd.
- A FOUND_MACHINE or NOT_FOUND message from a W-Schedd.

The following paragraphs describe the actions of the W-Startd when it is activated for one of these reasons.

time-out Periodically the W-Startd does the following things:

1. The W-Startd connects to the Collector and sends the GIVE_STATUS command, after which the Collector sends all the machine records to the W-Startd. The W-Startd makes a list of FREE_MACH records of the machines that are idle. A machine is considered to be idle if:
   - the START expression in the machine context is TRUE;
   - the State expression in the machine context is equal to "NoJob";

- the Machine expression in the machine context is unequal to "world".

This last test is necessary to prevent the World Machine from presenting itself as a real server to the other pools.

2. For every pool that has the right to run jobs in this pool, the W-Startd connects to the W-Startd of that pool and sends the command FREE_MACHINES and the list of FREE_MACH records.
3. The W-Startd chooses how it will present itself to the Central Manager. The W-Startd searches for a idle machine list (FREE_MACH_LIST record) with a large enough time_stamp (defined in the Condor configuration files). If there is such a list, the W-Startd copies the machine context of the first FREE_MACH record of this list. Otherwise it uses a standard context which indicates that the World Machine is unavailable. The W-Startd sends the command STARTD_INFO and the chosen context to the Collector.

FREE_MACHINES The W-Startd checks if the message comes from a World Machine of a pool where this pool has the right to run jobs. The message is ignored if this is not true. The W-Startd releases the records of the idle machine list of this pool, and then receives and stores the new FREE_MACH records. The time_stamp member of the FREE_MACH_LIST record of the pool is set to the current time.

GIVE_MACHINE The W-Startd receives the cluster_id/job_id pair and the job context from the Schedd. The W-Startd walks through the list of idle machines in search of a machine that both meets the job requirements and the job preferences. If no machine was found, the W-Startd walks a second time through the list of idle machines, this time for a machine that only meets the job requirements.

If the W-Startd has found a idle machine for the job, it sends the REQUEST_MACHINE command, the name of the requesting machine, the cluster_id/job_id pair, and the job context to the W-Schedd of the pool where the idle machine is located. The W-Startd creates and stores a SERVING record with information about this request.

If the W-Startd hasn't found a idle machine, it sends the command NO_MACHINE and the cluster_id/job_id pair to the requesting Schedd.

FOUND_MACHINE The W-Startd receives the name of the server, the name of the requesting machine, and the cluster_id/job_id pair from the W-Schedd. The W-Startd looks up the SERVING record for this request, and checks if the message comes from the expected World Machine. If so, the W-Startd sends the command MACHINE, the name of the server, and the cluster_id/job_id pair to the requesting Schedd. Otherwise the message is ignored.

NOT_FOUND The W-Startd receives the name of the requesting machine, and the cluster_id/job_id pair from the W-Schedd. The W-Startd checks if there is a SERVING record that corresponds with this message, and if the message comes from the W-Schedd of the pool from which it requested a machine. If this is true it sends the command

NO_MACHINE and the cluster_id/job_id pair to the requesting Schedd. Otherwise, the message is ignored.

6.5 The W-Schedd

The W-Schedd manages a data structure called the queue of requests. The records of this doubly-linked list are defined as:

```
typedef struct request {
        struct request  *next;
        struct request  *prev;
        char            *pool_name;
        char            *machine_name;
        PROC_ID         id;
        CONTEXT         *job_context;
        int             result;
        char            *server;
} REQUEST;
```

The member pool_name, machine_name, and id contain the name of the pool that posed the request, the name of the initiating machine, and the cluster_id/job_id pair of the job. The member job_context contains the job context consisting of expressions that describe the job requirements, job preferences and the job owner. The members result and server are used by the W-Schedd to temporarily store the results of the negotiations.

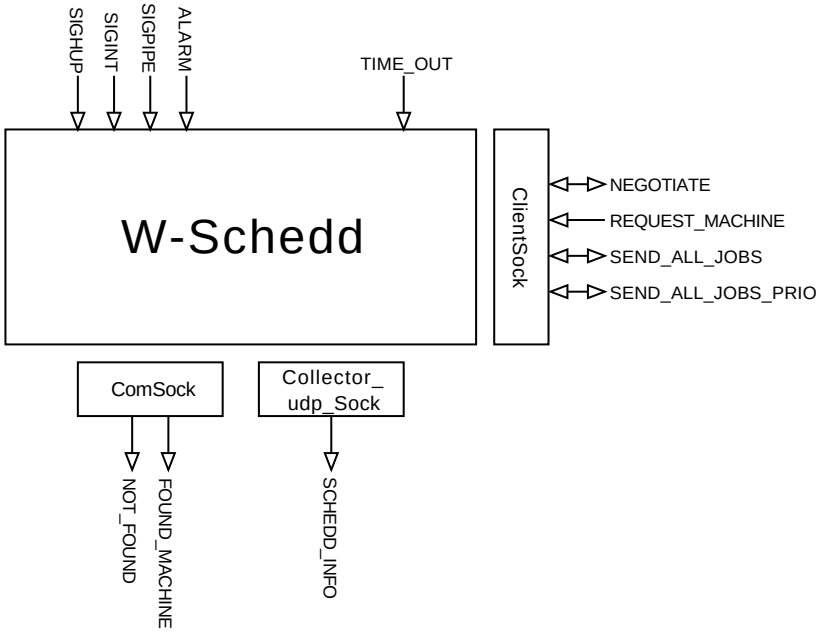


Figure 6.2 W-Schedd activation.

Figure 6.2 shows that the W-Schedd is activated when it receives the REQUEST_MACHINE command from a W-Startd, the NEGOTIATE command from the Collector, or when a certain interval (defined in the Condor configuration file) has elapsed. We now describe the actions the W-Schedd performs when it is activated.

time-out The W-Schedd creates a context with the following expressions:

Running = 0
Idle = [number of requests in the queue]
Users = 0

and sends the command SCHEDD_INFO and this context to the Collector. Running and Users are expressions used by the Negotiator to calculate the priority of a machine. They are set to zero so that the priority of the W-Schedd will always be at zero. The expression Idle is used by the Negotiator to determine whether a machine wants to run jobs. The W-Schedd sets this expression to the number of requests in the queue.

REQUEST_MACHINE The W-Schedd receives the name of the initiating machine, the cluster_id/job_id pair, and the job context, from the W-Startd and checks if the request came from a World Machine (Condor pool) that has the right to run jobs in this pool. If this is true, the W-Schedd makes a request record and places the request in the queue. The result member is initialized with the value REJECTED. If this is the first request in the queue, the W-Schedd sends a SCHEDD_INFO message to the Collector (see time-out), to tell that it wants to run a job.

NEGOTIATE The W-Schedd will negotiate with the Negotiator to get servers for the request in the queue. The W-Schedd puts the results of the negotiations in the queue, and sends the results to the W-Startd daemons of the pools that did the requests after the negotiations have ended. This is done to prevent that the Negotiator has to wait while the W-Schedd is communicating with the W-Startd.

The negotiations end when there are no more requests in the queue (the W-Schedd sends the Negotiator the NO_MORE_JOBS command), or when the World Machine has lost the priority (the Negotiator sends the END_NEGOTIATE command). The following list gives the steps of the negotiation for one job. This list is repeated until the negotiations end.

1. The Negotiator starts by sending the SEND_JOB_INFO command, or the END_NEGOTIATE command when the World Machine has lost the priority.
2. The W-Schedd replies by sending the JOB_INFO command and the job context of the next request, or when there are no more requests, the command NO_MORE_JOBS.
3. The Negotiator will try to find a server for the job. If the Negotiator has found a server, it sends the command PERMISSION and the name of the server to the W-Schedd, otherwise it sends the command REJECTED.

4. The W-Schedd puts the result (PERMISSION or REJECTED) in the result member, and the name of the server in the server member of the request record.

When the negotiations have ended, the W-Schedd goes through the queue and sends for every request the results to the W-Startd who did the request. If the result was PERMISSION, the W-Schedd sends the FOUND_MACHINE command, the name of the server, the name of the initiating machine, and the cluster_id/job_id pair, otherwise the command NOT_FOUND, the name of the initiating machine, and the cluster_id/job_id pair. After the W-Schedd has sent the result to the W-Startd, the request is deleted from the queue.

If the Negotiator ended the negotiations, the W-Schedd will send the NOT_FOUND message to the W-Startd daemons of the requests that where not handled, because the result member of the request records were initialized with the value REJECTED.

SEND_ALL_JOBS(_PRIO) The program condor_globalq asks all the machines that want to run jobs, to send information about these jobs. If the World Machine has told the Collector that it wants to run jobs, the program condor_globalq will send a SEND_ALL_JOBS or SEND_ALL_JOBS_PRIO message to the W-Schedd when it is run. The Schedd will respond by sending a empty PROC record, to indicate that the W-Schedd doesn't have a real job queue.

Chapter 7

Performance of Remote System Calls

We have tested the performance of the remote system call mechanism of Condor. We measured the costs of gettimeofday system calls, the costs of writing 1024 bytes to a file with a write system call, and the costs of reading 1024 bytes from a file with a read system call. We used a small job that measured the time needed to do 1000 write system calls for writing in total 1024000 bytes to an already existing file, the time needed to do 1000 gettimeofday system calls, and the time needed to read in the file of 1024000 bytes via 1000 read system calls.
We compared the following five situations:

- Without Condor. We measured the costs of normal system calls on several machines that are part of the NIKHEF pool.
- With Condor in the NIKHEF pool. We measured the costs of remote system calls for Condor jobs that were submitted in the NIKHEF pool (from paramount, the central file-server).
- With Condor in the flock situation with FWI. We measured the cost of remote system calls for Condor jobs that were submitted to a test flock consisting of a pool of three machines of NIKHEF and a pool of three machines of FWI.
- With Condor, execution machine is na47sun06. For measuring the costs of remote system calls between NIKHEF and CERN, we added a machine of the CERN pool to the NIKHEF pool. The jobs were submitted from paramount and executed on the machine na47sun06.
- With Condor, execution machine is morbier. As with the machine of CERN, we added the machine morbier of the WISCONSIN pool to the NIKHEF pool.

Although we performed the tests several times and at various points of time, the results only give an indication of the kind of delay that may be encountered. The machines of NIKHEF and the machines of FWI are connected by 10 Mbit ethernet. NIKHEF and FWI are both

connected to WCW-Lan, also a 10 Mbit ethernet. Between Amsterdam and CERN a 0.5 Mbit hired line is used, that is part of the EBONE network. The communication from Amsterdam to Wisconsin goes first to CERN, from where it goes over a 1.5 Mbit transatlantic line to Cornell. We have no information about the connection between Cornell and Wisconsin.

situation	best	worst	mean
without Condor	0.038	0.110	0.061
with Condor, NIKHEF pool	4.0	12.3	6.6
with Condor, NIKHEF→FWI	5.7	12.2	8.8
with Condor, NIKHEF→CERN	62	229	121
with Condor, NIKHEF→Wisconsin	164	1294	485

Table 7.1 Costs of gettimeofday system call in milliseconds.

Table 7.1 shows the costs in milliseconds of doing a gettimeofday system call. For example, it takes a Condor job in the NIKHEF pool on the average 6.6 milliseconds to do one gettimeofday remote system call. An important point shown by Table 7.1 is that there is only a small difference in performance between the situation where both the initiating and execution machine are situated in the NIKHEF pool, and the situation where the initiating machine is situated in the NIKHEF pool and the execution machine is situated in the FWI pool. This means that it is acceptable to use the remote system call mechanism in the Condor flock situation between NIKHEF and FWI.

situation	best		worst		mean	
	writing	reading	writing	reading	writing	reading
without Condor, without NFS—local disk	0.3	0.2	1.9	0.4	0.7	0.3
without Condor, with NFS	1.6	0.3	4.4	0.5	3.0	0.4
with Condor, NIKHEF pool, without NFS—remote system calls	7.6	7.2	15.5	17.2	10.7	11.1
with Condor, NIKHEF pool, with NFS	1.5	0.3	8.0	0.3	5.4	0.3
with Condor, NIKHEF→FWI	16.7	10.7	43.5	29.7	24.7	18.1
with Condor, NIKHEF→CERN	103	78	326	387	195	183
with Condor, NIKHEF→Wisconsin	212	213	1898	1552	643	582

Table 7.2 Costs of write and read system calls in milliseconds.

Table 7.2 shows the cost in milliseconds of write and read system calls respectively, each transferring 1024 bytes. We measured the performance of normal system calls and remote

system calls for jobs in the NIKHEF Condor pool, in both the situation that NFS was, or was not used. An important point shown by the information in Table 7.2 is that the NFS optimization is important. It also shows a difference in costs between writing and reading a file via NFS.

Chapter 8

Conclusions

The restriction that nothing might be changed to the Central Manager and as little as possible to the other Condor software has caused a few problems. It is possible to use the World Machine in a test situation, but the following changes to the Condor control software are required, in order to use the World Machine in a real situation.

- The World Machine has a priority that is calculated by the Central Manager in the same way as the priority of a normal machine. This means that the jobs of the World Machine may be run before the jobs of a normal machine that has a lower priority than the World Machine. The Negotiator should be changed so that the World Machine is always the last machine for whose jobs the Negotiator searches servers.

- The Negotiator should be changed, so that when it is searching for a server for a job, the World Machine is the last machine for which the Negotiator checks if the job can run on it. This to prevent that jobs are run on the World Machine while there are idle machines in the initiating pool.

- In the Condor flock situation it will be necessary to run the jobs under the UID and GID of "nobody" or under a special Condor flock UID and GID. The Schedd knows whether a job is going to be executed on a machine of the initiating pool, or on a machine of another pool. A possible implementation would be to transfer this information from the Schedd via the Shadow to the Startd and the Starter on the execution machine. The Starter can than run the user job under the appropriate UID and GID.

  Another possibility is that the Starter checks if the combination of login name, UID and GID of the owner of the job is in the password file. If the user is not known, the Starter runs the Condor job under the UID and GID of "nobody" or the special Condor flock UID and GID. If the user is known, the Starter runs the Condor job under the UID and GID of the owner. A problem with this approach is that the same combination of login name, UID and GID may exist in more than one pool.

  Some changes to the Condor system call stubs will be necessary, in order to prevent that files are accessed via NFS in the Condor flock situation. Otherwise an error can

occur when the Condor job tries to open a file via NFS, because the job doesn't run under the UID and GID of the owner, so it may not have the right to read or write the file.

There are some other points on which the design of the group flock can be improved. The following improvements are not essential for using the World Machine in a real situation.

- If a job from one pool is executed on a machines of another pool, the report returned by the program condor_status[1] gives an inconsistent view of the states of the Condor pools. In the initiating pool, the initiating machine says the job is running, but there is no corresponding machine that is serving the job. In the execution pool, the execution machine is serving a job (the machine is in the Job Running state) while there is no corresponding machine with a remotely executing job.

  The problem can be solved by letting the World Machine present itself to the Central Manager as a machine which is serving several jobs (the jobs of the pool that are executed on machines of other pools), and which has several jobs in the queue that are running (the number of jobs that other pools are executing on machines of the pool). The World Machine should then keep track of which jobs are running, and should be notified when a Condor job stops. A change to the Central Manager is needed to allow the World Machine to say that it is serving several Condor jobs.

- The W-Startd and W-Schedd can't be run on a machine that is part of the pool, because the W-Startd and W-Schedd use the same port numbers as the normal Startd and Schedd. It is possible to give the W-Startd and W-Schedd different port numbers than the normal Startd and Schedd, but this means that all the other Condor daemons have to know that the World Machine is a virtual machine.

In order to come to the ideal Condor flock situation as described in Section 4.3, it will be necessary to redesign the Central Manager. The World Machine will no longer present itself to the Central Manager as a machine with the same characteristics as one of the idle machines of the other pools, but it will provide the Central Manager with information about all the idle machines in the flock. The Negotiator can schedule the waiting jobs in the following way:

1. When the Negotiator starts scheduling it asks the status of the pool from the Collector, and information about the status of the flock from the World Machine. The information about the flock consists of a list of the idle machines in the other pools that may be used, and a list of received requests.

2. The Negotiator prioritizes the set of waiting Condor jobs (Condor jobs of the own pool and the requests from other pools), and determines the server-values of the idle machines.

[1] Even without flocking the information returned by condor_status may not be completely consistent (See Section 2.9).

3. The Negotiator can then allocate the idle machine with the highest server-value to the Condor job with the highest priority that can run on this machine.

Jobs that were scheduled to the World Machine can be started in the same way as described in Section 5.5 (the World Machine of the initiating pool requests an idle machine for the job from the World Machine of the execution pool). The following problems are solved by integrating the concept of the World Machine with the Central Manager.

- The assumption that all the machines in the flock have the same architecture and operating system is no longer necessary. The World Machine will provide the Central Manager with information about which idle machines may be used, and which characteristics they have.
- The Central Manager can schedule more than one job per schedule interval to the World Machine. The World Machine such as it has been implemented during my graduating term, can deal with the assignment of more than one job during one schedule interval.

Appendix A

Source Code W-Startd

Appendix B

Source Code W-Schedd

Appendix C

Changes to the Condor Schedd

Appendix D

Used Procedures from the Condor Libraries

dprintf_config(char *subsys, int logfd) Sets up the following dprintf variables based on the configuration file.

int DebugFlags	Bits to look for in dprintf
int MaxLog	Maximum size of the log file
char *DebugFile	Name of the log file
char *DebugLock	Name of the lock file
int (*DebugId)()	Function to call to print special info
FILE *DebufFP	The FILE to perform output to

dprintf(va_dcl va_alist) Generic logging function. Prints the message on DebugFP with a date if any bits in "flags" are set in DebugFlags. If locking is desired, DebugLock should contain the name of the lock file. If log length management is desired, MaxLog should contain the maximum length of the log in bytes. (The log will be copied to "DebugFile.old", so MaxLog should be half of the space you are willing to devote. If both log length management and locking are desired, the lock file should not be the same as the log file. Along with the date, other identifying information can be logged with the message by supplying the function (*DebugId)() which takes DebugFP as an argument.

EXCEPT(va_dcl va_alist) This is a macro that prints the line number, the name of the source file, and the error message (the argument of EXCEPT()) in the log file, and then exits.

config(char *a_out_name, CONTEXT *context) This routine reads in the Condor configuration files. The macros are stored in a hash table, and a context is created from the control functions.

char *param(char *name) Returns the value associated in the has table with the named parameter. Returns NULL if the given parameter is not defined.

do_connect(char *host, char *service, u_short port) Creates a stream socket and connects to the host-service or host-port combination.

udp_connect(char *host, u_short port) Creates a datagram socket and connects to the host-port combination.

XDR *xdr_Init(int *sock, XDR *xdrs) Creates a XDR record stream on top of a stream connection.

XDR *xdr_Udp_Init(int *sock, XDR *xdrs) Creates a XDR record stream on top of a datagram connection.

xdr_mywrapstring(XDR *xdrs, char **str) Decodes or encodes a string with use of primitive XDR routines. A null pointer is send as a null string, and a received null string is returned as a null pointer.

xdr_context(XDR *xdrs, CONTEXT *context) Decodes or encodes a context with use of primitive XDR routines.

xdr_mach_rec(XDR *xdrs, MACH_REC *ptr) Decodes or encodes a machine record with use of primitive XDR routines.

rcv_int(XDR *xdrs, CONTEXT *context, int end_of_record) This routine decodes an integer, and moves the input stream's position to the beginning of the next record in the stream if end_of_record is true.

rcv_string(XDR *xdrs, CONTEXT *context, int end_of_record) This routine decodes a string, and moves the input stream's position to the beginning of the next record in the stream if end_of_record is true.

snd_int(XDR *xdrs, CONTEXT *context, int end_of_record) This routine encodes an integer, and causes the outgoing data to be marked as a record if end_of_record is true.

snd_context(XDR *xdrs, CONTEXT *context, int end_of_record) This routine encodes a context, and causes the outgoing data to be marked as a record if end_of_record is true.

CONTEXT *create_context() Allocates memory for, and initializes a context data structure.

free_context(CONTEXT *context) Frees a context data structure.

store_stmt(EXPR *expr, CONTEXT *context) Adds an expression to a context data structure. If the expression already exists, it is replaced by the new one.

EXPR *scan(char *line) Parses a string and creates an expression.

EXPR *build_expr(char *name, ELEM *val) Makes an expression of the form "variable = value".

evaluate_bool(char *name, int *answer, CONTEXT *context)

evaluate_int(char *name, int *answer, CONTEXT *context)

evaluate_float(char *name, float *answer, CONTEXT *context)

evaluate_string(char *name, char **answer, CONTEXT *context) These four routines evaluate the value of the variable name, with use of expressions from context. The result is put in the variable answer.

Bibliography

[1] M.J. Bach, *The design of the UNIX operating system*, Prentice Hall, 1986.

[2] A. Bricker, M.J. Litzkow and M. Livny, "Condor technical summary," Version 4.1b, University of Wisconsin - Madison, 1991.

[3] A. Bricker and M.J. Litzkow, UNIX manual pages: condor_intro(1), condor(1), condor_q(1), condor_rm(1), condor_status(1), condor_summary(1), condor_config(5), condor_control(8), condor_master(8), Version 4.1b, University of Wisconsin - Madison, 1991.

[4] X. Evers, *A literature study on scheduling in distributed systems*, TU-Delft, October 1992.

[5] A. Goscinski and M. Bearman, "Resource management in large distributed systems," *Operating Systems Review*, Vol. 24, no. 4, 1990, pp. 7—25.

[6] A. Goscinski, *Distributed Operating Systems - The logical design*, Addison-Wesley publishing company, 1991.

[7] C. Hunt, *TCP/IP network administration*, Nutshell Series, O'Reilly & Associates, Inc., 1992.

[8] B.W. Kernighan and D.M. Ritchie, *The C programming language*, second edition, Prentice Hall, 1988.

[9] L. Lamport, LaTeX*: A document preparation system*, Addison-Wesley publishing company, 1986.

[10] L. Lamport, LaTeX*local guide for Nikhef*, revised by Marcel Prins and Gertjan Stil, 1987.

[11] M.J. Litzkow, "Remote UNIX, turning idle workstations into cycle servers," in *Proceedings of the 1987 Summer Usenix Conference*, Phoenix, Arizona, 1987.

[12] M.J. Litzkow, M. Livny and M.W. Mutka, "Condor - A hunter of idle workstations," in *Proceedings of the 8th International Conference on Distributed Computing Systems*, San Jose, California, 1988, pp. 104—111.

[13] M.J. Litzkow and M. Livny, "Experience with the CONDOR distributed batch system," *Proceedings of the IEEE Workshop on Experimental Distributed Systems*, Huntsville, AL, 1990.

[14] M.J. Litzkow and M. Solomon, "Supporting checkpointing and process migration outside the UNIX kernel," *Usenix Winter Conference*, San Francisco, California, 1992.

[15] M.W. Mutka and M. Livny, "Scheduling remote processing capacity in a workstation-processor bank network," in *Proc. 7th Int. Conf. Dist. Comp. Systems*, Berlin, West Germany, 1987, pp. 2—9.

[16] M.W. Mutka and M. Livny, "Profiling workstations' available capacity for remote execution," in *Proceedings of Performance '87, The 12th IFIP W.G. 7.3 International Symposium on Computer Performance Modeling, Measurement and Evaluation*, Brussels, Belgium, 1987, pp. 529—544.

[17] S. Talbott, *Managing projects with make*, Nutshell Series, O'Reilly & Associates, Inc., 1986.