A Worldwide Flock of Condors: Load Sharing among Workstation Clusters

D.H.J. Epema[a], M. Livny[b], R. van Dantzig[c], X. Evers[a,c], and J. Pruyne[b]

[a] *Department of Mathematics and Computer Science, Delft University of Technology, P.O. Box 356, 2600 AJ Delft, The Netherlands*

[b] *Department of Computer Sciences, University of Wisconsin–Madison, Madison, Wisconsin, USA*

[c] *National Institute for Nuclear Physics and High-Energy Physics Research (NIKHEF), P.O. Box 41882, 1009 DB Amsterdam, The Netherlands*

Abstract

Condor is a distributed batch system for sharing the workload of compute-intensive jobs in a pool of UNIX workstations connected by a network. In such a Condor pool, idle machines are spotted by Condor and allocated to queued jobs, thus putting otherwise unutilized capacity to efficient use. When institutions owning Condor pools cooperate, they may wish to exploit the joint capacity of their pools in a similar way. So the need arises to extend the Condor load-sharing and protection mechanisms beyond the boundaries of Condor pools, or in other words, to create a *flock* of Condors. Such a flock may include Condor pools connected by local-area networks as well as by wide-area networks.

In this paper we describe the design and implementation of a distributed, layered Condor flocking mechanism. The main concept in this design is the *Gateway Machine* that represents in each pool idle machines from other pools in the flock and allows job transfers across pool boundaries. Our flocking design is transparent to the workstation owners, to the users, and to Condor itself. We also discuss our experiences with an intercontinental Condor flock.

Keywords: distributed processing, batch queueing system, wide-area load sharing, ownership rights, flocking.

1 Introduction

Today, many organizations use personal workstations as a computing platform. These workstations are typically connected by a local-area network (LAN), and—for large organizations—sometimes by a wide-area network (WAN). In most cases, each workstation is placed at the disposal of an employee, the workstation *owner*, who has full control over its resources. Together, these workstations represent a substantial computing resource and a significant investment for the organization. In recent years, it has become apparent that these ever more powerful workstations can support a large class of computational problems. Additionally, because they are manufactured in large quantities, they are in most cases more cost-effective than mainframes.

The distributed nature with respect to ownership and location of these clusters of workstations requires a new approach to the way computing resources are allocated within an organization. While the problem of managing resources that are physically distributed has been addressed by many researchers, the distributed-ownership aspect of clusters of desk-top workstations is new. It has been observed [12] that most owners have computation needs that are much smaller than the capacity of their workstations and therefore tend to leave them idle for long periods of time. At the same time, a small group of owners who belong to the same organization may have batch-mode computing needs that are by far larger than what their workstations can provide. It is therefore not uncommon to find a cluster of workstations in the undesirable *wait-while-idle* (WWI) state [9], in which batch jobs are waiting while elsewhere resources capable of serving them are idle.

Any attempt to increase the amount of resources available for batch processing on a cluster of workstations (and thus reduce the time spent in the WWI state) must first of all guarantee the rights of each and every workstation owner. While the ultimate goal of a batch system is to make essentially the *entire* computing power of the cluster available for batch processing, it is the owner of the workstation who has the right to decide when and by whom the workstation can be used for batch processing. Condor [7] is the first batch system for clusters of workstations to address the distributed-ownership problem. It provides owners with means to control the impact batch processing has on the quality of service they experience on their workstation. Other batch systems, such as DQS [3], LSF [17], LoadLeveler [5] (that is based on Condor), and Codine [2], have also recently made such means available. Since Condor became operational in 1988, it has been proven [8] that turning a cluster of workstations into a *Condor pool* results in a substantial increase in productivity and efficiency. In a wide range of industrial and academic settings, Condor has demonstrated its ability to put the capacity of large clusters of workstations to efficient use serving the needs of demanding interactive owners and batch users.

While the formation of Condor pools solves the WWI problem within an organization, it does not address this problem across organizations. When groups, departments or institutes with Condor pools have a mutual interest in the progress of their computational activities, they are likely to notice that the rate at which their computation tasks are accomplished is hindered by a WWI problem across their Condor pools. Fluctuations in workloads, time-zone differences, and different working habits are likely to result in situations where one pool is overloaded while other pools are underutilized. Although the design of Condor does not preclude the merging of different pools into one pool with WAN connections, Condor was not designed to protect the rights of an organization to its own cluster. As in the case of an individual who owns a workstation, an organization that owns a Condor pool would be willing to make unutilized resources available to other organizations only if its ownership rights to these resources are fully protected.

In this paper we present a mechanism that enables a controlled exchange of computing resources across the boundaries of Condor pools. Using this so-called *flocking* mechanism, independent Condor pools can be turned into a *Condor flock* where jobs submitted in one pool—the *submission* pool—may access resources belonging to another pool—the *execution* pool. In such a flock, any two pools can be connected by a pair of Gateway (GW) machines—one in either pool. These GW machines serve as resource brokers between the two pools, and take care of the transfer of jobs. GW machines behave like any other workstation in a pool—they advertise resources, they accept jobs for execution, and they request computing resources for jobs. The only difference is that the resources they advertise and the jobs whose needs they are trying to satisfy reside in a different Condor pool. Each pool owner and each workstation owner maintains full control on when their resources can be used by external jobs. The access to resources across pool boundaries is under control of the GW machines, and is transparent to the workstation owners, to the users, and to Condor.

In 1993, a prototype of the Condor flocking mechanism was designed and implemented as the master's thesis project for the Delft University of Technology of one of the authors [4], carried out at NIKHEF, as part of an informal collaboration among the authors' institutions. In 1994, the flock software was installed at various institutes and experience was obtained in an intercontinental flock, including Condor pools in Madison (USA), Amsterdam and Delft (The Netherlands), Geneva (Switzerland), Warsaw (Poland) and Dubna (Russia). Thereafter the flocking mechanism was adapted and upgraded by the Condor Group at the University of Wisconsin–Madison to its present form, which is compatible with the latest version of Condor.

The remainder of this paper is organized as follows. A summary of Condor is given in Section 2. In Section 3, the main design alternatives concerning Condor flocking are discussed, and in Section 4, the design and the implementation of the flocking mechanism are presented.

Section 5 contains a brief account of our experiences with Condor flocking, and a short discussion of areas for further improvement of our flocking mechanism. In Section 6 we present our conclusions.

2 The Condor System

Condor is the result of more than a decade of research and development at the Computer Sciences Department of the University of Wisconsin–Madison. In this section we summarize those aspects of the Condor system which are relevant for this paper. A detailed description of the system can be found in [1, 6, 7]. The following three principles have guided the design of Condor.

1. Condor batch processing should have almost no impact on the availability of and the quality of service provided by workstations to their owners.

2. Condor should be fully responsible for locating the resources required by a batch job, letting the job use these resources, monitoring its execution, and informing the user on its progress.

3. Condor should not require special programming and should preserve the operating environment of the machine on which a job was submitted.

We consider design principle 1 as essential, since when it is not satisfied, owners do not allow their workstations to be part of a Condor pool.

2.1 How Condor Works

Each workstation in a Condor pool runs two daemons, the scheduler daemon *Schedd* and the starter daemon *Startd*. One of the workstations in the pool is designated as the *Central Manager* (CM), and runs some daemon processes for this purpose. The Startd of a workstation periodically advertises to the CM the resources of the workstation, encapsulated in a *machine context*, and whether it is available (idle). In addition, the Startd starts, monitors, and terminates jobs that were assigned to the workstation by the CM. The Schedd queues jobs submitted to Condor at the workstation and seeks resources for them. Each job has a *job context* defining its resource requirements. This job context is forwarded to the CM, who tries to locate a workstation that meets these requirements. The CM can be viewed as a matchmaker, matching job contexts and machine contexts. The CM performs scheduling by scanning its list of queued jobs for potential matches in an order based on a novel priority scheme [10] in which jobs are ranked according to the past resource-usage pattern of the user who submitted them.

We now present the protocol used by Condor for matching a job J queued on *submission* machine S and an *execution* machine E, and for the subsequent starting of the job (see Figure 1). Both machines are in the same Condor pool and we refer to this protocol as the standard Condor protocol. In the protocol, processes on machines S and E are denoted by their name with "(S)" and "(E)" appended, respectively.

The standard Condor protocol

1. Matchmaking:
   (a) The Schedd(S) sends J's job context to the CM[1]
   (b) The Startd(E) sends E's machine context to the CM
   (c) The CM identifies a match between J's requirements and E's resources
   (d) The CM sends the Schedd(S) an identification of E
2. Establishing a connection:
   (a) The Schedd(S) forks a Shadow(S)
   (b) The Shadow(S) passes the Startd(E) J's job requirements
   (c) If the Startd(E) finds that J and E indeed match and that E is still idle, it sends an OK message containing an identification of E to the Shadow(S). If either of these conditions is not satisfied, it sends a not-OK message, and steps 1 and 2 are repeated
3. Starting the job:
   (a) The Startd(E) forks a Starter(E)
   (b) The Shadow(S) sends J's executable to the Starter(E)
   (c) The Starter(E) forks J

The function of the Shadow process is to represent the Condor job on the submission machine (see Subsection 2.2), while the function of the Starter process is to represent the job on the execution machine.

2.2 Remote System Calls

A Condor job runs on an execution machine with the illusion that it operates in the environment of the submission machine. This illusion is maintained by the Remote UNIX facility [6], through which a number of system calls (which mainly have to do with file I/O) are redirected to

[1] Steps 1.(a) and 1.(b) can be executed in any order or in parallel.

the submission machine, where they are handled by the Shadow. This *remote-system-call* mechanism is implemented in a special version of the C-library. For each remote system call, the Condor C-library contains a stub. The stub traps the system call and forwards it to the Shadow. When the Shadow receives the call, it executes the corresponding system call and sends the result back to the stub. Then the stub returns to the application program in exactly the same way the normal system call would have. To obtain this service the user only has to link his jobs with the Condor version of the standard C-library.

When a Condor pool has a uniform file system like NFS or AFS, the file-I/O system calls can be handled by the execution machine directly. When no uniform file system is available, or when checkpointing (see Subsection 2.3) is needed, some system calls, such as those for process creation and interprocess communication are not allowed by Condor; they are then trapped by the Shadow.

2.3 Job Preemption

Condor operates in an environment where owners have absolute priority on their machines. In order to ensure this, Condor can preempt jobs to free a machine for its owner's use. Preemption is also induced by the CM in order to revoke resources from low-priority users to be allocated to high-priority users. In either event, the machine running the job must be vacated quickly, cleaned up, and prepared for its owner or a new Condor job. At the same time, there needs to be some guarantee that jobs make progress in spite of these preemptions. Condor, therefore, provides mechanisms that automatically checkpoint [15] and restart a job (usually on a different machine). After a job has been checkpointed, its submission machine requests the CM to reschedule the job with the standard protocol presented above.

3 Design Alternatives

Our basic objective is to solve the *inter*pool WWI problem among the Condor pools of cooperating owners, given that the *intra*pool WWI problem has been solved by Condor. This means that users can get access to more or different resources than the ones available in their local pools. Since Condor allows WAN connections and does not require a uniform file system or a uniform userid/password environment, the simplest way to solve the WWI problem among pools is to *merge* them into one large pool with a single CM. Another way to address the intrapool WWI problem is to *connect* the Condor pools by means of a *flocking* mechanism.

Before evaluating these two alternatives, we would like to list the design principles that have guided us in solving the intrapool WWI problem. Our starting point is that the solution should be transparent to both the workstation owners and the users. Therefore, the Condor design principles 1-3 stated in Section 2 remain valid. We identify the following three additional design principles, all of which are related to pool ownership and management:

4. The installation and maintenance of any additional mechanisms must be easy;
5. Adding a pool to a set of cooperating pools, maintaining such a set, and withdrawing a pool from such a set, must be easy and fast;
6. It must be possible to define simple and flexible resource-sharing agreements between the pool owners.

Although it is the users who are to benefit from a solution to the WWI problem, it is truly indispensable to have the cooperation of first the workstation owners—as is the case for a single Condor pool—and then the pool owners.

Below we first introduce the concept of resource-sharing agreements between pool owners that allow a Condor flock to be a dynamic entity. Then we demonstrate that simply merging pools conflicts with our design principles, so that a flocking mechanism is needed. Finally, we discuss two issues regarding the structure of flocks.

3.1 Resource-Sharing Agreements between Pool Owners

Among a number of cooperating owners of Condor pools, not every one of them has to cooperate with every other. When two owners do cooperate, they may want to set rules specifying the circumstances under which job transfers between their pools are allowed. These rules may for example depend on the sizes of the pools, the nature of their workloads, or the nature of the cooperation. We refer to the set of rules that govern the exchange of jobs between two pools as a (*resource-sharing*) *agreement*. Widely different kinds of rules are possible, for example:

- Any idle machine in one pool can be used by any Condor job of any user in the other pool. This is an agreement that is similar to that which usually exists between the owners of workstations within a single Condor pool.
- Job transfer can be restricted. For instance, a pool owner may require that the number of idle machines in his pool must exceed a certain threshold before transfers to it are allowed; job transfers may only be allowed in one direction between two pools; jobs of only specific users of one pool may be allowed in the other pool, or may be required to have a lower priority in the other pool than local Condor jobs.
- Rules may include some form of accounting. For instance, it may be required that in the long run, either pool uses roughly the same amount of resources in the other pool.

3.2 Merging Pools versus Flocking

The basic question is whether there should be flocking at all. Why not just merge the pools of organizations wishing to eliminate the WWI problem among their Condor pools into one common pool? We now discuss the advantages and disadvantages of merging and flocking with respect to our design principles 1-6. As to design principles 1-3 stated in Subsection 2.1, obviously merging will not violate them, and it seems easy to have a flocking mechanism that follows them. So we can concentrate on the last three design principles (4-6) stated at the beginning of this section.

4. In a merged pool, no additional mechanisms have to be installed at all. However, one of the constituent pools has to be designated to contain the single CM. This means that the installation of (new versions of) the CM and the responsibility for keeping the CM (and so the entire merged pool) running, is with only one of the participating organizations. This does not seem to be an attractive proposition.

5. Adding the Condor pool of an organization to and withdrawing it from a merged pool entail substantial changes to the configuration file in the CM, which may reside in another organization. A flocking mechanism, on the other hand, allows a Condor pool to be instantaneously (re-)connected to and disconnected from any pool in the flock with which it has an agreement. Also, in a merged pool, any change in one of the constituent pools must be communicated to the administrator of the merged pool, while in a flock, no communication on such changes is necessary.

   Obviously, in a merged pool, the single CM forms a potential performance bottleneck. On the other hand, a flocking mechanism may achieve some level of failure isolation: while in a single, merged pool, any failure of *the* CM stops all job-resource matching activity, in a flock, a failure of the CM of *a* pool leaves the rest of the flock unaffected.

6. All users of a Condor pool tend to have equal rights. In case of a merged pool, therefore, the users of an organization will in principle not have a special claim to their own machines. Although some form of priority can be enforced in a single Condor pool, maintaining the pertaining information is cumbersome. On the other hand, a properly designed flocking mechanism will allow organizations to keep full authority over their own machines. They can choose and change the Condor configuration parameters themselves, and facilities can be included for admitting and refusing jobs from other pools selectively by means of flexible, bilateral agreements.

We conclude that it is unlikely that cooperating pool owners are willing to merge their Condor pools. Therefore, a flocking mechanism is called for.

3.3 Central versus Distributed Flock Structure

Whenever computers are combined to serve a common purpose, the question arises whether the authority in the combined system should be central or distributed. Here we are faced with this question for the authority in Condor flocks.

In principle, one can build a flock in a hierarchical way as a pool of pools, with a single *Flock Central Manager* (FCM) presiding over the CMs of all the pools, and being responsible for all interpool scheduling decisions. The FCM should be aware of the pools that belong to the flock, their configurations, and their resource-sharing agreements. Periodically, the FCM should collect from all CMs dynamic status information regarding resource availability and job requests, it should make matches, and it should send its resource-allocation decisions back to the CMs involved. It would then be the responsibility of the CMs to carry out these interpool scheduling decisions, along with their own intrapool scheduling decisions. The most important advantage of making decisions centrally is simplicity: pools have to register only at the FCM, and all information exchange concerning the flock only involves the CMs and the FCM.

A disadvantage of the central approach is that the FCM has still to be installed and maintained by a single organization. Another disadvantage is that when more and more pools are added—and we envision truly global flocks consisting of a very large number of pools—the FCM and its local network may become overloaded. In addition, an FCM only provides an intermediate form of failure isolation. A failure of the FCM results in a collapse of the entire flock-management system, although the individual pools can continue to operate correctly. An advantage of the central approach is that it allows resource-sharing agreements by simply implementing them at the FCM level.

In a distributed structure, the decision making can be distributed among the pools by having any pair of Condor pools negotiate the transfer of jobs without any interference from other pools. In order for a pool to become a participant in a flock, it should be able to *connect* to a subset of the pools in the flock, without necessarily affecting or even being aware of the others. Making a connection between pools should include the implementation of the bilateral agreement of their owners. Any change in the agreement between two pool owners only entails an information exchange between the two pools involved. In such a way, a distributed flock can be tailored to become a self-controlled and self-growing entity. We conclude that the distributed approach is superior to the central approach.

3.4 Integrated versus Layered Design

We distinguish two ways in which a flocking mechanism can be added to Condor. In the first, such a mechanism is *integrated* into the Central Managers. This means that the CMs have to exchange information regarding job submissions and machine availability, and that they decide for a locally submitted job whether to transfer it or to process it locally. In the second way, a separate software *layer* is put on top of Condor, taking scheduling from the intrapool level to the interpool level.

The main advantage of an integrated design is that the CMs can use any local and non-local information to optimize their scheduling decisions. When flocking is completely integrated in the CMs and the CMs are all equal, an integrated design will necessarily be distributed, which we consider to be an advantage.

The main advantage of a layered design is that the flocking mechanism can be developed independently from standard Condor. In particular, it may even allow the CM to remain unmodified. This enables an orthogonality of design not possible in the integrated approach, it facilitates installation and testing of the flocking mechanism, and it makes the decision to join a flock easier for a pool owner. A disadvantage of such a design is that it may prevent certain optimizations. We conclude that a layered design may stimulate the success of flocking considerably.

4 The Design and Implementation of a Condor Flock

In this section we describe our design of a distributed, layered flocking mechanism for Condor. We first give a description of the flock structure. Then we discuss the protocol for starting a Condor job in another pool. We conclude the section with some details on the way scheduling is performed through the GWs. Our flocking mechanism is transparent to the workstation owners, the users, the CMs, and the Condor daemons on the workstations.

4.1 The Gateway Machines

The basis of our flocking mechanism is formed by the *Gateway Machines*—at least one in every participating pool. The purpose of these GW machines is to act as resource brokers between pools. To this end, they include facilities for connecting pools, for exchanging information regarding resource availability, and for binding jobs to external resources. We now discuss these three facilities in turn.

A pool may contain multiple GWs, and a GW may be used to connect a pool to several pools. Each GW has a *flock configuration file* describing the subset of connections maintained by the GW. For each of these connections, this file contains the name of the pool and the network address of the GW at the other end, and whether the local pool is allowed to run Condor jobs in the remote pool and vice versa. Connecting two pools is done by entering the appropriate information in the flock configuration files in a GW in either pool. By changing the flock configuration file of one of its GWs, a pool can connect to or disconnect from other pools, and can dynamically change its agreements with the pools to which it is connected. Figure 2 shows a Condor flock consisting of three Condor pools.

Similarly as an ordinary Condor machine, a GW runs two daemons, the *GW-Schedd* and the *GW-Startd*. Periodically, the GW-Startds exchange information on the availability of machines in their pools. To this end, a GW-Startd requests the status of its pool from the CM using the interface of the Condor utility program allowing users to query the status of a pool. From the information received, the GW-Startd makes a list of available machines in the pool. It then sends this list to the GW-Startds of all GWs to which it is connected that belong to pools that have permission to run jobs in this pool. If a GW-Startd does not receive a list of available machines from another pool within a specified time, the latter is considered to be down, and the previous list is deleted. Note that the information exchange only involves the availability of resources, not job submissions.

The protocol between a GW machine and the CM in its pool is identical to that between an ordinary machine and the CM. Periodically, the GW-Startd chooses a machine from the availability lists received, and presents the GW to the CM with the characteristics of this machine (for details, see Subsection 4.3). If a GW does not have information on idle machines in remote pools, it presents itself as a machine that is unavailable. It is now possible that the CM makes a match between a local job and the (machine represented by the) GW. The steps involved in establishing the connection between the job and the actual execution machine are given in the following subsection.

4.2 How Flocking Works

In our design, the CM of a pool that is part of a Condor flock does not know anything about flocking. Therefore, step 1 of the standard Condor protocol given in Subsection 2.1 is performed in exactly the same way as in a single pool. (When E is a GW, E's machine context and resources are those of a remote machine rather than those of the GW.) When the execution machine E is an ordinary machine, the rest of the standard protocol is followed. When a match is made between a job J on submission machine S and a GW machine in the submission pool, step 2 in the standard protocol is replaced by steps 2.I-2.IV below. We extend the notation of Subsection 2.1 by denoting processes on the GW in the submission pool and on the GW in the execution pool to which it is connected, by their name with "(S)" and "(E)" appended, respectively. Figure 3 illustrates the entire protocol for starting a job in a different pool.

The Condor Flocking protocol (step 2)

2.I Establishing a connection between S and E:

(a) The Schedd(S) forks a Shadow(S)

(b) The Shadow(S) passes the GW-Startd(S) J's requirements

(c) The GW-Startd(S) forks a GW-Startd-child(S)

(d) The GW-Startd-child(S) passes the GW-Schedd(E) J's job context

2.II Matchmaking *within the execution pool*:

This step is identical to step 1 of the standard Condor protocol with E a machine in the execution pool and Schedd(S) replaced by GW-Schedd(E)

2.III Establishing a connection *within the execution pool*:

This step is identical to step 2 of the standard Condor protocol with Schedd(S) replaced by GW-Schedd(E) and with Shadow(S) replaced by GW-Simulate-Shadow(E)

2.IV Completing the connection:

(a) Depending on the answer obtained by the GW-Simulate-Shadow(E) in step 2.III, it sends the GW-Startd-child(S) an OK message and an identification of the actual execution machine E, or a not-OK message

(b) GW-Startd-child(S) sends an OK message containing an identification of E, or a not-OK message to the Shadow(S)

After this new step 2 has been executed, the submission machine and the execution machine are in the same states as in the standard protocol, and the GWs are no longer involved. Step 3 of the Condor flocking protocol is identical to step 3 of the standard protocol. All subsequent communication between the submission machine and the execution machine for remote system calls and checkpointing are done just as they would be in a single Condor pool, i.e., with no additional overhead.

The complete protocol can be summarized as follows. First, in the submission pool a match is made between a job and a (pseudo-)execution machine, a GW; then the job is queued in a GW in the execution pool; then a match is made in the execution pool; and finally, the id of the actual execution machine is communicated to the submission machine. An important aspect of the protocol is that the Schedd and the Startd operate in exactly the same way as in the current version of standard Condor, so flocking is transparent to the ordinary machines in a Condor pool.

All the messages concerned with starting a Condor job in a remote pool contain the name of the submission machine and the job id. This is necessary because the GW daemons may be working for several jobs submitted on different machines and in different pools at the same time.

Normally in a Condor pool, a job runs under the user identifier (UID) of the user who submitted the job, so that files can be accessed through AFS or NFS. In a Condor flock, a user of one pool will usually not be registered in the other pools. Therefore, we let transferred jobs run under the special UID "nobody".

4.3 Scheduling Details

Since in our design flocking is transparent to the CMs, all interpool scheduling and all resource-sharing agreements have to be implemented by algorithms executed by the GWs. In particular, it is up to the GW to choose a machine from the availability lists received from other pools to represent itself to the CM, and to decide to which pool to send a job allocated to it. In our design, this is done in the following way.

Whenever a GW needs to represent a machine to the CM, it chooses at random a machine of the availability list from a randomly chosen pool to which it is connected. If all the availability lists are empty, the GW will represent itself as a machine that is unavailable. This procedure is repeated periodically in order to give the CM more opportunities to make matches, and also when a job is assigned to the GW. Note that because the GW does not know the requirements of the submitted jobs, it cannot represent the machine that is in some way the best. When a job is assigned to the GW, it will not necessarily be executed on the machine advertised by the GW, and it may even be sent to an execution pool different from the one containing the advertised machine. What happens is that the GW scans the availability lists in random order until it encounters a machine satisfying the job requirements and the job preferences. It then sends the job to the pool to which this machine belongs. If no such machine is found, this procedure is repeated to find a pool with an idle machine which only satisfies the job requirements. If still no machine is found, the job remains queued at the submission machine and has to be rescheduled.

When a flocked job is checkpointed in the execution pool, its checkpoint file is sent back to the submission machine. When it is later rescheduled, it may be flocked again, or it may be assigned to a machine in the submission pool. So a job may use resources in any number of pools in a flock during its lifetime.

Finally, a Condor user can indicate for each of his jobs whether it is to run in the submission pool, or whether it can be flocked. In this detail, flocking is not necessarily transparent to the users.

5 Experiences and Future Developments

In 1993 a first test flock was set up with three small Condor pools using the prototype implementation. These pools were owned by NIKHEF, the Faculty of Mathematics and Computer Science of the University of Amsterdam (FWI-UA), and the Department of Mathematics and Computer Science of Delft University of Technology, all in The Netherlands. Each of the pools consisted of 3 Sun SPARC workstations.

In the summer of 1994, a production flock of nine pools in five different countries (the USA, The Netherlands, Switzerland, Poland and Russia) was set up in a collaboration of the authors' institutions with the Spin Muon Collaboration (SMC) at CERN, the Institute of Computing and Automation (LCTA) of the Joint Institute for Nuclear Research (JINR) in Dubna near Moscow, and the Physics Department of Warsaw University. This first "World Flock" contained over 250 workstations, about 60 of which on the European continent (see Figure 4). The flock was made available as a production environment for the research project "Crystallization on a Sphere" [16], in which a very large number of time-consuming simulated annealing (SA) jobs for a variable number of particles (N) were executed. Thousands of these production jobs were submitted—usually in batches of 100 jobs—in the various pools and executed remotely in the flock. The duration of the jobs depended strongly on the value of N. The remote execution times ranged from 15 minutes to 10 days per job. In addition, some simulations of high-energy collisions using the GEANT package developed at CERN were used as tests. All of these jobs had low I/O requirements, which was important since some connections were slow. The feasibility of our flocking approach was solidly demonstrated. There were no significant differences in execution efficiency between SA production jobs executed in their submission pool and those that were flocked.

At the end of 1994, version 5 of Condor was released by the Condor Design Team, with an accompanying upgrade of the flocking package. This version of Condor has been in routine operation in two large heterogeneous pools at the University of Wisconson–Madison (in the Departments of Computer Sciences and of Computing and Engineering), and in smaller pools elsewhere, amongst which pools at NIKHEF and at the University of Delft. Towards the end of 1995, a new flock is emerging based on these latest versions of Condor and the flocking package. Our experience with flocking so far convinced us that the chosen flocking mechanism meets our design principles, and that it can have a significant impact on the computing facilities available to participating organizations.

Our current implementation still lacks various facilities that we consider to be important for broad application. Two of these are discussed below. As both have to do with performance, we are currently building monitoring tools for Condor flocks to identify bottlenecks and areas for performance improvements.

I/O Issues

Pools in a Condor flock will often be connected by WAN connections. Therefore, an efficiency aspect of flocking is the speed of transfer across such connections of executables, checkpoint files, and I/O data. The suitability of jobs to be flocked strongly depends on the ratio of I/O requirements and the bandwidth. In the spectrum of jobs submitted in a Condor pool, there may well be a significant fraction of jobs for which this ratio is low enough to benefit substantially from flocking. To get a feeling for the average and variation in remote-system-call delays, we performed various measurements using single pools with and without WAN connections. After all, once the connection between the submission and execution machines has been established either within a single pool or across pool boundaries in a flock, these machines are in the same states. We compared the performance of remote system calls in various instances of three different situations: within a single machine, between two machines in a single LAN-connected pool, and between a machine in the NIKHEF pool and a machine at CERN and at the University of Wisconsin–Madison (both across a WAN connection). The measurements, showing considerable fluctuations, indicated that bandwidth limitations are not prohibitive for WAN-flocking in the case of compute-intensive jobs.

Scheduling Issues

The main objective of Condor—making available idle computing cycles while protecting ownership rights—is quite different from the traditional objectives of scheduling such as optimal average turn-around time. Still, there are various scheduling issues to be resolved in a Condor flock, concerning optimization and stability. Optimization issues include the choice of the jobs to be transferred and the resources to be allocated to them. For example, one might want to match the job with the longest expected execution time and the fastest available machine, or the job with the highest I/O requirements and a slower machine. Also, the question arises whether a job should be flocked immediately when submitted, or whether one should first wait to see whether a local machine turns idle. In our design, any such policies can be implemented in the GW machines. Stability issues concern the avoidance of unnecessary transfers. For example, when local capacity is available, it should be put to use. In particular, it should usually be avoided that among two pools, many jobs of one of them are flocked to the other and vice versa at the same time.

6 Conclusions

A design and implementation of Condor flocking, a novel concept in large-scale cluster computing, has been presented. The flocking mechanism has been tested, and the feasibility of both LAN-connected flocks and of worldwide flocks has been demonstrated over a period of several months. Obviously, the basic flocking concepts are not restricted to Condor, and not even to UNIX, but can be applied among any type of computing facilities. We envision long-running, compute-intensive jobs submitted to global heterogeneous flocks, successively being allocated resources in different locations worldwide to make progress in their execution.

New challenges in scientific computing, for example in experimental and theoretical physics, which include simulations of orders of magnitude larger than those currently performed, can benefit greatly from these concepts. Also, scientists who are severely restricted in their access to computing resources, could profit from the nocturnal tidal wave of computing power of idle workstations traveling around the globe every 24 hours. We conclude that flocking has the potential of becoming an illuminating example of global sharing of otherwise wasted resources.

7 Acknowledgements

We are grateful to M. Litzkow (University of Wisconson) and R. Boontje (NIKHEF) for their technical advice during the design of Condor flocking. We gladly acknowledge the excellent work done by Dhrubajyoti Borthakur, who maintained the 1994 World Flock and upgraded the software while integrating it in version 5 of Condor. R. Boer (Delft University/NIKHEF) deserves thanks for his comments on drafts of this paper, the description of the protocols, and for making the figures, M. Litmaatd (NIKHEF) for his ideas and help in the early stages of this work. Finally, we thank our colleagues who by their cooperation and active support have contributed to the work reported here, in particular M.K. Ballintijn and B. Segal (CN/CERN, Geneva), W. Heubers, P.U ten Kate (NIKHEF), J.F.C.M. de Jongh (Delft University), M. Popov and R. Pose (JINR, Dubna), W. Bogusz, M. Gromisz and J. Polec (Warsaw University), and P.M.A. Sloot, J. Vesseur and J. Voogd (FWI, University of Amsterdam). Part of this work was financially supported by the Foundation for Fundamental Research on Matter (FOM) and the Netherlands Organization for Scientific Research (NWO).

References

[1] A. Bricker, M.J. Litzkow and M. Livny, Condor Technical Summary, Version 4.1b, Technical Report 1069, Computer Sciences Department, University of Wisconsin–Madison, Wisconsin, USA (1992).

[2] Codine, Computing in Distributed Networked Environments, User's Guide and Reference Manual, Genias Software GmbH, Erzgebirgstr. 2B, D-93073 Neutraubling, Germany (1994).

[3] D. Duke, T. Green, and J. Pasko, Research towards a heterogeneous networked computing cluster: The distributed queueing system version 3.0, Technical Report, Supercomputer Computations Research Institute, Florida State University (1994).

[4] X. Evers, Condor Flocking: Load Sharing between Pools of Workstations, Master's Thesis, Department of Mathematics and Computer Science, Delft University of Technology (1993).

[5] IBM LoadLeveler: User's Guide, Doc. No. SH26-7226-00, IBM Corporation (1993).

[6] M.J. Litzkow, Remote UNIX, turning idle workstations into cycle servers, Proc. of the 1987 Usenix Summer Conference, Phoenix, Arizona, USA (1987) 381-384.

[7] M.J. Litzkow, M. Livny and M.W. Mutka, Condor–A hunter of idle workstations, Proc. of the 8th Int'l Conf. on Distributed Computing Systems, San Jose, CA, USA (1988) 104-111.

[8] M.J. Litzkow and M. Livny, Experience with the Condor distributed batch system, Proc. of the IEEE Workshop on Experimental Distributed Systems, Huntsville, AL, USA (1990).

[9] M. Livny and M. Melman, Load Balancing in Homogeneous Broadcast Distributed Systems, Proc. of the ACM Computer Network Performance Symposium, College Park, Maryland (1982).

[10] M.W. Mutka and M. Livny, Scheduling remote processing capacity in a workstation-processor bank network, Proc. of the 7th Int'l Conf. on Distributed Computing Systems, Berlin, Germany (1987) 2-9.

[11] M.W. Mutka and M. Livny, Profiling workstations' available capacity for remote execution, Performance '87, Proc. of the 12th IFIP WG 7.3 Int'l Symp. on Computer Performance Modeling, Measurement and Evaluation, Brussels, Belgium (1987) 529-544.

[12] M.W. Mutka and M. Livny, The available capacity of a privately owned workstation environment, *Performance Evaluation* Vol. 12 (1991) 269-284.

[13] J. Pruyne and M. Livny, Providing resource management services to parallel applications, in: J. Dongarra and B. Tourancheau, eds., Proc. of the Second Workshop on Environments and Tools for Parallel Scientific Computing, SIAM Proceedings Series (SIAM, 1994) 152-161.

[14] J. Pruyne and M. Livny, Parallel processing on dynamic resources with CARMI, in: D. G. Feitelson and L. Rudolph, eds., Job Scheduling Strategies for Parallel Processing, Lecture Notes in Computer Science Vol. 949 (Springer-Verlag, 1995) 259-278.

[15] T. Tannenbaum and M.J. Litzkow, The Condor Distributed Processing System, *Dr Dobbs Journal* Vol. 20(2) (1995) 40-48.

[16] J.M. Voogd, P.M.A. Sloot and R. van Dantzig, Crystallization on a sphere, *FGCS* Vol. 10 (1994) 359-361 (and references therein).

[17] S. Zhou, J. Wang, X. Zheng, and P. Delisle, Utopia: A load sharing facility for large, heterogeneous distributed computing systems, Technical Report CSRI-257, Computer Systems Research Institute, University of Toronto (1992).

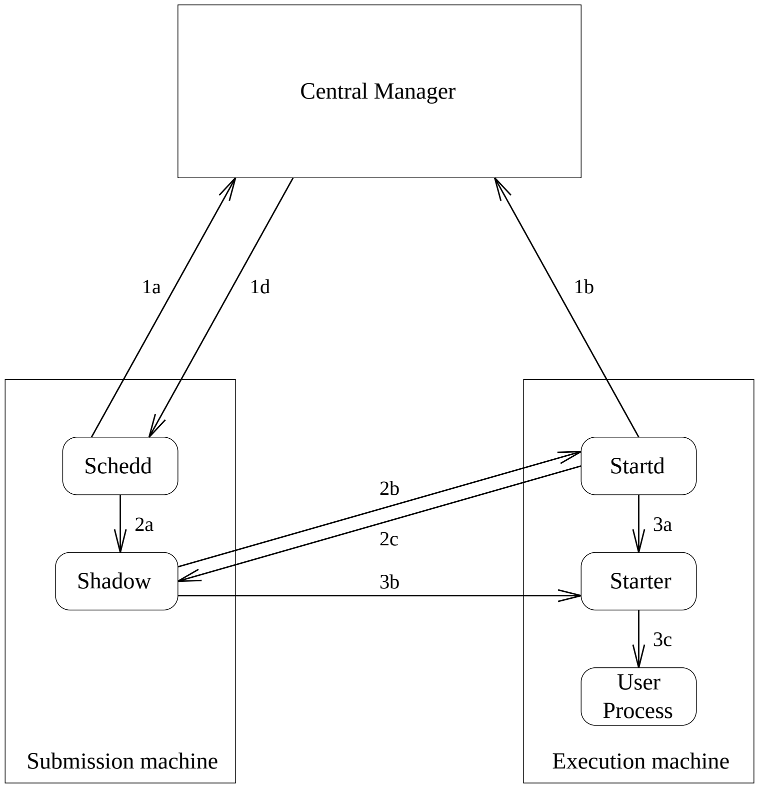

Figure 1: The standard Condor protocol. The arrows and labels correspond to the steps of the protocol in Subsection 2.1.

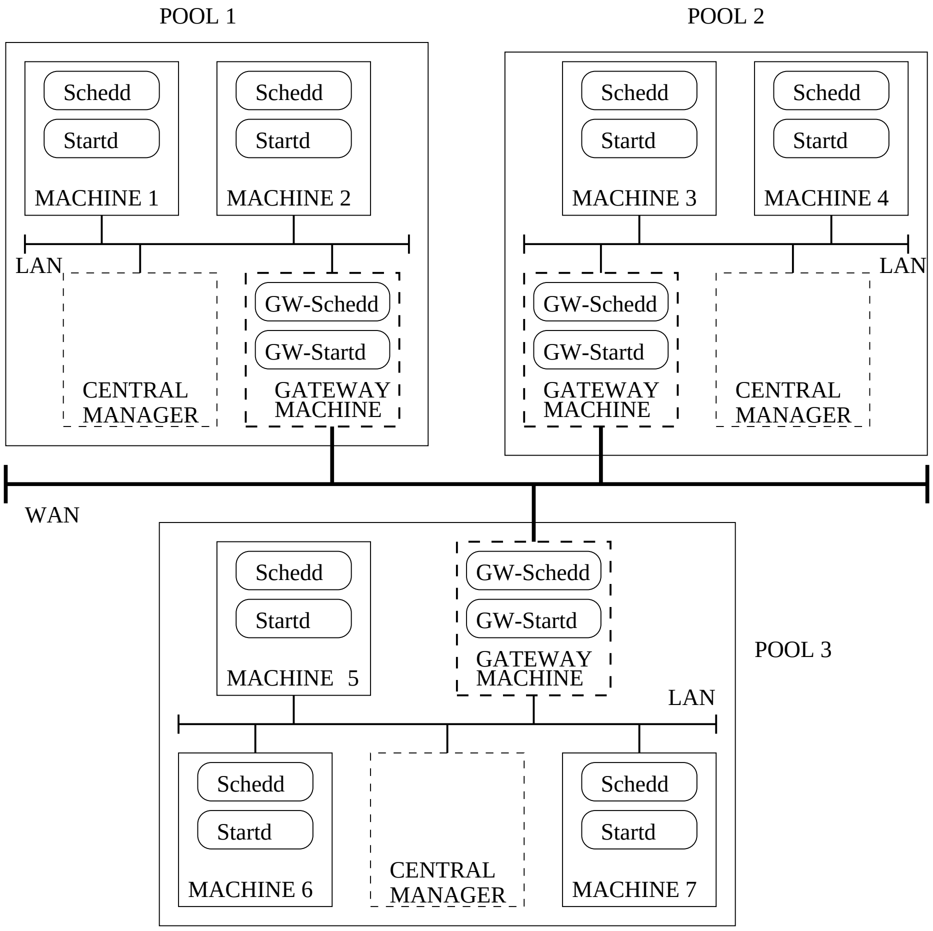

Figure 2: An example of a Condor flock with three pools connected by a wide-area network. The pools consist of two or three machines each, connected by a local-area network.

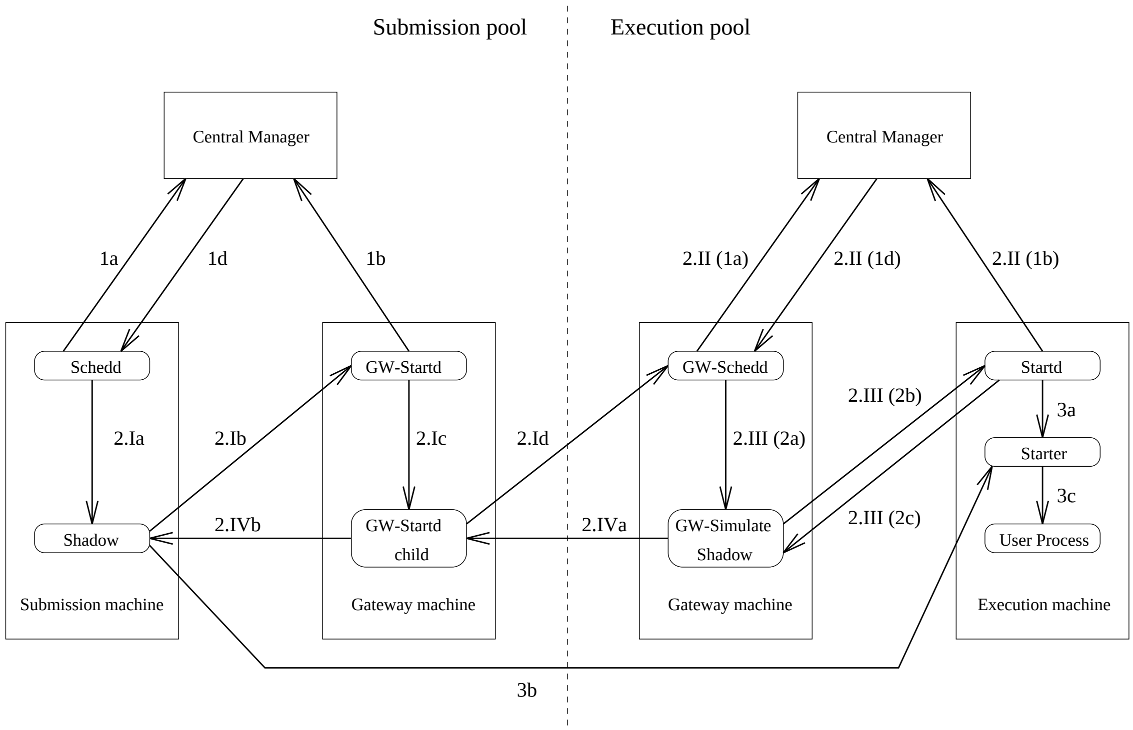

Figure 3: The Condor Flocking protocol. The arrows and their labels correspond to the steps of the protocol in Subsection 4.2, and of the standard Condor protocol in Subsection 2.1. The labels in parentheses correspond to the steps of the standard Condor protocol used within steps of the Condor Flocking protocol.

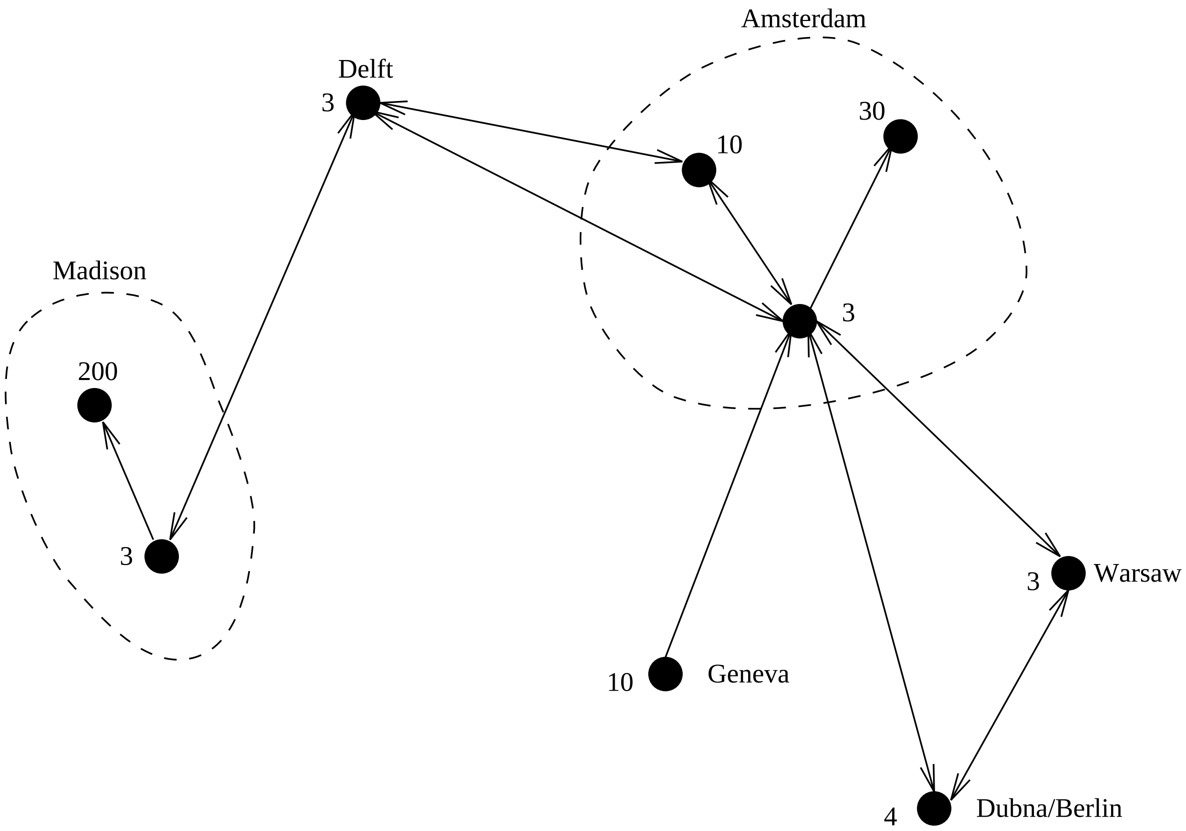

Figure 4: The world-wide Condor flock of 1994. The dots represent the pools. An arrow from one pool to another indicates that jobs submitted in the former are allowed in the latter. The numbers indicate the sizes of the pools.