Condor and the Grid

Douglas Thain, Todd Tannenbaum, and Miron Livny

Computer Sciences Department, University of Wisconsin-Madison
1210 West Dayton Street, Madison WI 53706

SUMMARY

Since 1984, the Condor project has helped ordinary users to do extraordinary computing. Today, the project continues to explore the social and technical problems of cooperative computing on scales ranging from the desktop to the world-wide computational grid. In this chapter, we provide the history and philosophy of the Condor project and describe how it has interacted with other projects and evolved along with the field of distributed computing. We outline the core components of the Condor system and describe how the technology of computing must reflect the sociology of communities. Throughout, we reflect on the lessons of experience and chart the course travelled by research ideas as they grow into production systems.

KEY WORDS: Condor, grid, history, community, planning, scheduling, split execution

Introduction

> *Since the early days of mankind the primary motivation for the establishment of communities has been the idea that by being part of an organized group the capabilities of an individual are improved. The great progress in the area of inter-computer communication led to the development of means by which stand-alone processing subsystems can be integrated into multi-computer communities.*
>
> - Miron Livny, "Study of Load Balancing Algorithms for Decentralized Distributed Processing Systems," Ph.D. thesis, July 1983.

Ready access to large amounts of computing power has been a persistent goal of computer scientists for decades. Since the 1960s, visions of computing utilities as pervasive and as simple as the telephone have driven system designers. [57] It was recognized in the 1970s that such power could be achieved inexpensively with collections of small devices rather than expensive single supercomputers. Interest in schemes for managing distributed processors [69, 26, 23] became so popular that there was even once a minor controversy over the meaning of the word "distributed." [30]

As this early work made it clear that distributed computing was *feasible*, theoretical researchers began to notice that distributed computing would be *difficult*. When messages may be lost, corrupted, or delayed, precise algorithms must be used in order to build an understandable (if not controllable) system. [46, 45, 24, 56] Such lessons were not lost on the system designers of the early 1980s. Production systems such as Locus [78] and Grapevine [21] recognized the fundamental tension between consistency and availability in the face of failures.

In this environment, the Condor project was born. At the University of Wisconsin, Miron Livny combined his 1983 doctoral thesis on cooperative processing [53] with the powerful Crystal Multicomputer [29] designed by Dewitt, Finkel, and Solomon and the novel Remote UNIX [52] software designed by Litzkow. The result was Condor, a new system for distributed computing. In contrast to the dominant centralized control model of the day, Condor was unique in its insistence that every participant in the system remain free to contribute as much or as little as it cared to.

> *Modern processing environments that consist of large collections of workstations interconnected by high capacity network raise the following challenging question: can we satisfy the needs of users who need extra capacity without lowering the quality of service experienced by the owners of under utilized workstations? ... The Condor scheduling system is our answer to this question.*
>
> - Michael Litzkow, Miron Livny, and Matt Mutka, "Condor: A Hunter of Idle Workstations," IEEE 8th Intl. Conf. on Dist. Comp. Sys., June 1988.

The Condor system soon became a staple of the production computing environment at the University of Wisconsin, partially because of its concern for protecting individual interests. [50] A production setting can be both a curse and a blessing: The Condor project learned hard lessons as it gained real users. It was soon discovered that inconvenienced machine owners would quickly withdraw from the community, so it was decreed that owners must maintain control of their machines at any cost. A fixed schema for representing users and machines was in constant change and so led to the development of a schema-free resource allocation language called ClassAds. [62, 63, 61] It has been observed [48] that most complex systems struggle through an adolescence of five to seven years. Condor was no exception.

> *The most critical support task is responding to those owners of machines who feel that Condor is in some way interfering with their own use of their machine. Such complaints must be answered both promptly and diplomatically. Workstation owners are not used to the concept of somebody else using their machine while they are away and are in general suspicious of any new software installed on their system.*
>
> - Michael Litzkow and Miron Livny, "Experience With The Condor Distributed Batch System," IEEE Workshop on Experimental Dist. Sys., October 1990.

The 1990s saw tremendous growth in the field of distributed computing. Scientific interests began to recognize that coupled commodity machines were significantly less expensive than supercomputers of equivalent power [67]. A wide variety of powerful batch execution systems such as LoadLeveler [27] (a descendant of Condor), LSF [81], Maui [43], NQE [42], and PBS [41] spread throughout academia and business. Several high profile distributed computing efforts such as SETI@Home and Napster raised the public consciousness about the power of distributed computing, generating not a little moral and legal controversy along the way [14, 68]. A vision called grid computing began to build the case for resource sharing across organizational boundaries [37].

Throughout this period, the Condor project immersed itself in the problems of production users. As new programming environments such as PVM [59], MPI [79], and Java [76] became popular, the project added system support and contributed to standards development. As scientists grouped themselves into international computing efforts such as the Grid Physics Network [4] and the Particle Physics Data Grid [10], the Condor project took part from initial design to end-user support. As new protocols such as GRAM [28], GSI [34], and GridFTP [13] developed, the project applied them to production systems and suggested changes based on the experience. Through the years, the Condor project adapted computing structures to fit changing human communities.

Many previous publications about Condor have described in fine detail the features of the system. In this chapter, we will lay out a broad history of the Condor project and its design philosophy. We will describe how this philosophy has led to an organic growth of *computing communities* and discuss the *planning* and *scheduling* techniques needed in such an uncontrolled system. Our insistence on dividing responsibility has led to a unique model of cooperative computing called *split execution*. We will conclude by describing how real users have put Condor to work.

The Philosophy of Flexibility

As distributed systems scale to ever larger sizes, they become more and more difficult to control or even to describe. International distributed systems are heterogeneous in every way: they are composed of many types and brands of hardware; they run various operating systems and applications; they are connected by unreliable networks; they change configuration constantly as old components become obsolete and new components are powered on. Most importantly, they have many owners, each with private policies and requirements that control their participation in the community.

Flexibility is the key to surviving in such a hostile environment. Five admonitions outline our philosophy of flexibility.

Let communities grow naturally. Humanity has a natural desire to work together on common problems. Given tools of sufficient power, people will organize the computing structures that they need. However, human relationships are complex. People invest their time and resources into many communities with varying degrees. Trust is rarely complete or symmetric. Communities and contracts are never formalized with the same level of precision as computer code. Relationships and requirements change over time. Thus, we aim to build structures that permit but do not require cooperation. Relationships, obligations, and schemata will develop according to user necessity.

Plan without being picky. Progress requires optimism. In a community of sufficient size, there will always be idle resources available to do work. But, there will also always be resources that are slow, misconfigured, disconnected, or broken. An over-dependence on the correct operation of any remote device is a recipe for disaster. As we design software, we must spend more time contemplating the consequences of failure than the potential benefits of success. When failures come our way, we must be prepared to retry or reassign work as the situation permits.

Leave the owner in control. To attract the maximum number of participants in a community, the barriers to participation must be low. Users will not donate their property to the common good unless they maintain some control over how it is used. Therefor, we must be careful to provide tools for the owner of a resource to set use policies and even instantly retract it for private use.

Lend and borrow. The Condor project has developed a large body of expertise in distributed resource management. Countless other practitioners in the field are experts in related fields such as networking, databases, programming languages, and security. The Condor project aims to give the research community the benefits of our expertise while accepting and integrating knowledge and software from other sources.

Understand previous research. We must always be vigilant to understand and apply previous research in computer science. Our field has developed over many decades, known by many overlapping names such as operating systems, distributed computing, meta-computing, peer-to-peer computing, and grid computing. Each of these emphasizes a particular aspect of the discipline, but are united by fundamental concepts. If we fail to understand and apply previous research, we will at best rediscover well-charted shores. At worst, we will wreck ourselves on well-charted rocks.

The Condor Project Today

At present, the Condor project consists of over 30 faculty, full time staff, graduate and undergraduate students working at the University of Wisconsin-Madison. Together the group has over a century of experience in distributed computing concepts and practices, systems programming and design, and software engineering.

Condor is a multi-faceted project engaged in five primary activities.

Research in distributed computing. Our research focus areas and the tools we have produced, several of which will be explored below, are:

- Harnessing the power of opportunistic and dedicated resources. *(Condor)*
- Job management services for grid applications. *(Condor-G, DaPSched)*
- Fabric management services for grid resources. *(Condor, GlideIn, NeST)*
- Resource discovery, monitoring, and management. *(ClassAds, Hawkeye)*
- Problem solving environments. *(MW, DAGMan)*
- Distributed I/O technology. *(Bypass, PFS, Kangaroo, NeST)*

Participation in the scientific community. Condor participates in national and international grid research, development, and deployment efforts. The actual development and deployment activities of the Condor project are a critical ingredient towards its success. Condor is actively involved in efforts such as the Grid Physics Network (GriPhyn) [4], the International Virtual Data Grid Laboratory (IVDGL) [6], the Particle Physics Data Grid (PPDG) [10], the

NSF Middleware Initiative (NMI) [9], TeraGrid [12], and the NASA Information Power Grid (IPG). [7] Further, Condor is a founding member in the National Computational Science Alliance (NCSA) [8] and a close collaborator of the Globus project. [36]

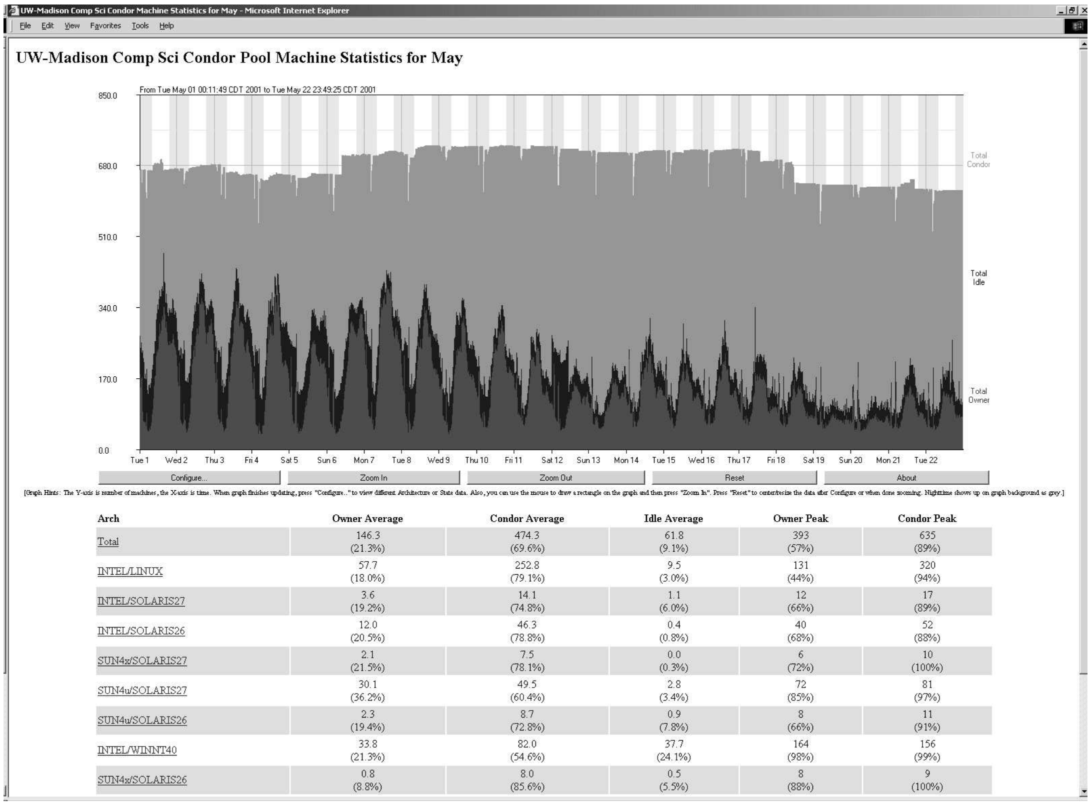
Engineering of complex software. Although a research project, Condor has a significant software production component. Our software is routinely used in mission critical settings by industry, government, and academia. As a result, a portion of the project resembles a software company. Condor is built every day on multiple platforms, and an automated regression test suite containing over 200 tests stresses the current release candidate each night. The project's code base itself contains nearly a half-million lines, and significant pieces are closely tied to the underlying operating system. Two versions of the software, a stable version and a development version, are simultaneously developed in a multi-platform (Unix and Windows) environment. Within a given stable version, only bug fixes to the code base are permitted — new functionality must first mature and prove itself within the development series. Our release procedure makes use of multiple testbeds. Early development releases run on test pools consisting of about a dozen machines; later in the development cycle, release candidates run on the production UW-Madison pool with over 1,000 machines and dozens of real users. Final release candidates are installed at collaborator sites and carefully monitored. The goal is each stable version release of Condor has been proven to operate in the field before being made available to the public.

Maintenance of production environments. The Condor project is also responsible for the Condor installation in the Computer Science Department at the University of Wisconsin-Madison, which consist of over 1000 CPUs. This installation is also a major compute resource for the Alliance Partners for Advanced Computational Servers (PACS) [2]. As such, it delivers compute cycles to scientists across the nation who have been granted computational resources by the National Science Foundation. In addition, the project provides consulting and support for other Condor installations at the University and around the world. Best effort support from the Condor software developers is available at no charge via ticket-tracked email. Institutions using Condor can also opt for contracted support — for a fee, the Condor project will provide priority email and telephone support with guaranteed turn-around times.

Education of students. Last but not least, the Condor project trains students to become computer scientists. Part of this education is immersion in a production system. Students graduate with the rare experience of having nurtured software from the chalkboard all the way to the end user. In addition, students participate in the academic community by designing, performing, writing, and presenting original research. At the time of this writing, the project employs twenty graduate students including seven Ph.D. candidates.

The Condor Software: Condor and Condor-G

When most people hear the word "Condor", they do not think of the research group and all of its surrounding activities. Instead, usually what comes to mind is strictly the *software* produced by the Condor project: the *Condor High Throughput Computing System*, often referred to simply as Condor.



Arch	Owner Average	Condor Average	Idle Average	Owner Peak	Condor Peak
Total	146.3 (21.3%)	474.3 (69.6%)	61.8 (9.1%)	393 (57%)	635 (89%)
INTEL/LINUX	57.7 (18.0%)	252.8 (79.1%)	9.5 (3.0%)	131 (44%)	320 (94%)
INTEL/SOLARIS27	3.6 (19.2%)	14.1 (74.8%)	1.1 (6.0%)	12 (66%)	17 (89%)
INTEL/SOLARIS26	12.0 (20.5%)	46.3 (78.8%)	0.4 (0.8%)	40 (68%)	52 (88%)
SUN4x/SOLARIS27	2.1 (21.5%)	7.5 (78.1%)	0.0 (0.3%)	6 (72%)	10 (100%)
SUN4u/SOLARIS27	30.1 (36.2%)	49.5 (60.4%)	2.8 (3.4%)	72 (85%)	81 (97%)
SUN4u/SOLARIS26	2.3 (19.4%)	8.7 (72.8%)	0.9 (7.8%)	8 (66%)	11 (91%)
INTEL/WINNT40	33.8 (21.3%)	82.0 (54.6%)	37.7 (24.1%)	164 (98%)	156 (99%)
SUN4x/SOLARIS26	0.8 (8.8%)	8.0 (85.6%)	0.5 (5.5%)	8 (88%)	9 (100%)

Figure 1. **Screenshot of CondorView** *CondorView software displays resource and customer utilization statistics for a given Condor pool via an interactive web interface. On a typical day, one Condor pool at UW-Madison delivers more than 650 CPU* days *(nearly two years of compute time) to UW researchers.*

Condor: A System for High Throughput Computing

Condor is a specialized job and *resource management system* (RMS) [33] for compute-intensive jobs. Like other full-featured systems, Condor provides a job management mechanism, scheduling policy, priority scheme, resource monitoring, and resource management. [70, 71] Users submit their jobs to Condor, and Condor subsequently chooses when and where to run them based upon a policy, monitors their progress, and ultimately informs the user upon completion.

While providing functionality similar to that of a more traditional batch queueing system, Condor's novel architecture and unique mechanisms allow it to perform well in environments where a traditional RMS is weak — areas such as sustained *high-throughput computing* and *opportunistic computing*. The goal of a high-throughput computing environment [17] is to provide large amounts of fault tolerant computational power over prolonged periods of time by effectively utilizing all resources available to the network. The goal of opportunistic computing is the ability to utilize resources whenever they are available, *without requiring* one hundred percent availability. The two goals are naturally coupled. High-throughput computing is most easily achieved through opportunistic means.

Some of the enabling mechanisms of Condor include:

- **ClassAds.** The ClassAd mechanism in Condor provides an extremely flexible and expressive framework for matching resource requests (e.g. jobs) with resource offers (e.g. machines). ClassAds allow Condor to adopt to nearly any desired resource utilization policy, and to adopt a *planning* approach when incorporating grid resources. We will discuss this approach further in a section below.
- **Job Checkpoint and Migration.** With certain types of jobs, Condor can transparently record a checkpoint and subsequently resume the application from the checkpoint file. A periodic checkpoint provides a form of fault tolerance and safeguards the accumulated computation time of a job. A checkpoint also permits a job to migrate from one machine to another machine, enabling Condor to perform low-penalty *preemptive-resume scheduling*. [44]
- **Remote System Calls.** When running jobs on remote machines, Condor can often preserve the local execution environment via remote system calls. Remote system calls is one of Condor's *mobile sandbox* mechanisms for redirecting all of a jobs I/O related system calls back to the machine which submitted the job. Therefore users do not need to make data files available on remote workstations before Condor executes their programs there, even in the absence of a shared filesystem.

With these mechanisms, Condor can do more than effectively manage dedicated compute clusters. [70, 71] Condor can also scavenge and manage wasted CPU power from otherwise idle desktop workstations across an entire organization with minimal effort. For example, Condor can be configured to run jobs on desktop workstations only when the keyboard and cpu are idle. If a job is running on a workstation when the user returns and hits a key, Condor can migrate the job to a different workstation and resume the job right where it left off.

Moreover, these same mechanisms enable preemptive-resume scheduling of dedicated compute cluster resources. This allows Condor to cleanly support priority-based scheduling on clusters. When any node in a dedicated cluster is not scheduled to run a job, Condor can utilize that node in an opportunistic manner — but when a schedule reservation requires that node again in the future, Condor can preempt any opportunistic computing job which may have been placed there in the meantime. [79] The end result: Condor is used to seamlessly combine all of an organization's computational power into one resource.

The first version of Condor was installed as a production system in the UW-Madison Department of Computer Sciences in 1987. [52] Today, in our department alone, Condor manages more than 1000 desktop workstation and compute cluster CPUs. It has become a critical tool for UW researchers. Hundreds of organizations in industry, government, and academia are successfully using Condor to establish compute environments ranging in size from a handful to thousands of workstations.

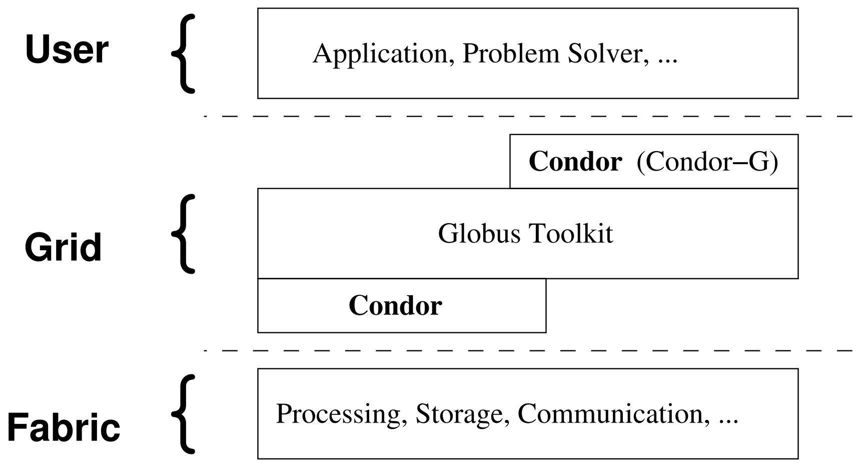


Figure 2. **Condor technologies in grid middleware** *Grid middleware consisting of technologies from both Condor and Globus sit between the user's environment and the actual fabric (resources).*

Condor-G: A Computation Management Agent for Grid Computing

Condor-G [38] represents the marriage of technologies from the Globus and Condor projects. From Globus [35] comes the use of protocols for secure inter-domain communications and standardized access to a variety of remote batch systems. From Condor comes the user concerns of job submission, job allocation, error recovery, and creation of a friendly execution environment. The result is very beneficial for the end user, who is now enabled to utilize large collections of resources which span across multiple domains as if they all belonged to the personal domain of the user.

Condor technology can exist at both the front and back ends of a middleware environment, as depicted in Figure 2. Condor-G can be used as the reliable submission and job management service for one or more sites, the Condor High Throughput Computing system can be used as the fabric management service (a grid "generator") for one or more sites, and finally Globus Toolkit services can be used as the bridge between them. In fact, Figure 2 can serve as a simplified diagram for many emerging grids, such as the USCMS Testbed Grid [3], established for the purpose of high-energy physics event reconstruction. Another example is the European Union DataGrid [1] project's Grid Resource Broker, which utilizes Condor-G as its job submission service [32].

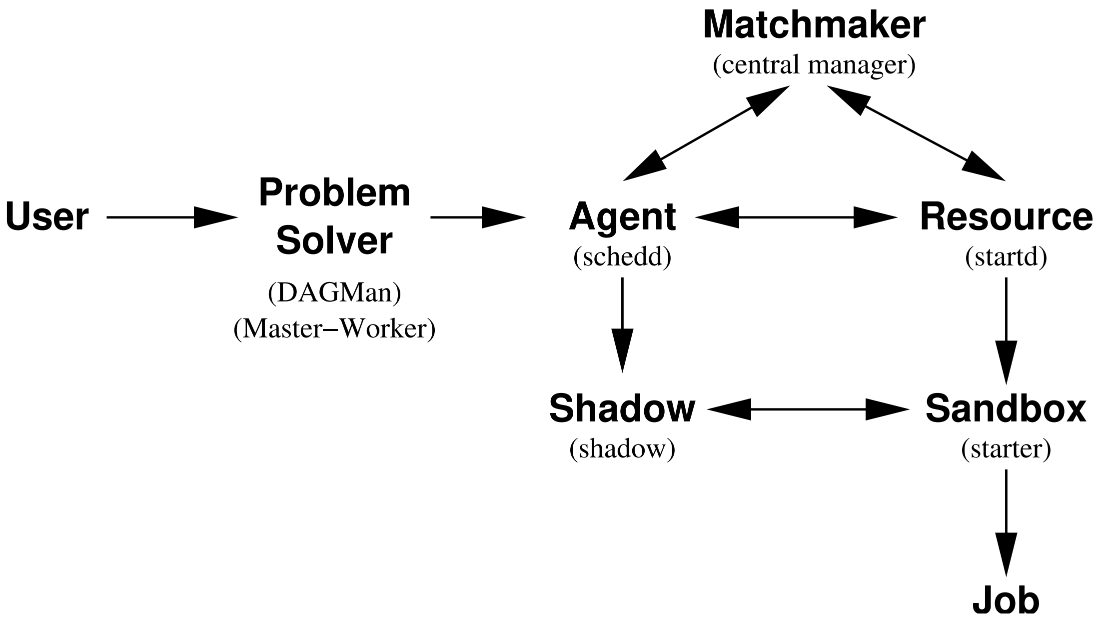


Figure 3. **The Condor Kernel** *This figure shows the major processes in a Condor system. The common generic name for each process is given in large print. In parentheses are the technical Condor-specific names used in some publications.*

A History of Computing Communities

Over the history of the Condor project, the fundamental structure of the system has remained constant while its power and functionality has steadily grown. The core components, known as the *kernel*, are shown in Figure 3. In this section, we will examine how a wide variety of *computing communities* may be constructed with small variations to the kernel.

Briefly, the kernel works as follows: The user submits jobs to an *agent*. The agent is responsible for remembering jobs in persistent storage while finding *resources* willing to run them. Agents and resources advertise themselves to a *matchmaker*, which is responsible for introducing potentially compatible agents and resources. Once introduced, an agent is responsible for contacting a resource and verifying that the match is still valid. To actually execute a job, each side must start a new process. At the agent, a *shadow* is responsible for providing all of the details necessary to execute a job. At the resource, a *sandbox* is responsible for creating a safe execution environment for the job and protecting the resource from any mischief.

Let us begin by examining how agents, resources, and matchmakers come together to form *Condor pools*. Later in this chapter, we will return to examine the other components of the kernel.

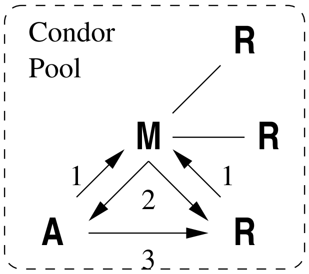


Figure 4. **A Condor Pool ca. 1988** *An agent (A) is shown executing a job on a resource (R) with the help of a matchmaker (M). Step 1: The agent and the resource advertise themselves to the matchmaker. Step 2: The matchmaker informs the two parties that they are potentially compatible. Step 3: The agent contacts the resource and executes a job.*

The initial conception of Condor is shown in Figure 4. Agents and resources independently report information about themselves to a well-known matchmaker, which then makes the same information available to the community. A single machine typically runs both an agent and a resource daemon and is capable of submitting and executing jobs. However, agents and resources are logically distinct. A single machine may run either or both, reflecting the needs of its owner. Furthermore, a machine may run more than one instance of an agent. Each user sharing a single machine could, for instance, run their own personal agent. This functionality is enabled by the agent implementation, which does not use any fixed IP port numbers or require any superuser privileges.

Each of the three parties — agents, resources, and matchmakers — are independent and individually responsible for enforcing their owner's policies. The agent enforces the submitting user's policies on what resources are trusted and suitable for running jobs. The resource enforces the machine owner's policies on what users are to be trusted and serviced. The matchmaker is responsible for enforcing community policies such as admission control. It may choose to admit or reject participants entirely based upon their names or addresses and may also set global limits such as the fraction of the pool allocable to any one agent. Each participant is autonomous, but the community as a single entity is defined by the common selection of a matchmaker.

As the Condor software developed, pools began to sprout up around the world. In the original design, it was very easy to accomplish resource sharing in the context of one community. A participant merely had to get in touch with a single matchmaker to consume or provide resources. However, a user could only participate in one community: that defined by a matchmaker. Users began to express their need to share across organizational boundaries.

This observation led to the development of *gateway flocking* in 1994. [31] At that time, there were several hundred workstations at Wisconsin, while tens of workstations were scattered across several organizations in Europe. Combining all of the machines into one Condor pool was not a possibility because each organization wished to retain existing community policies

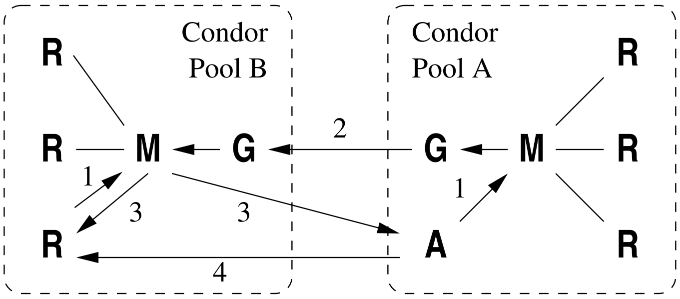


Figure 5. **Gateway Flocking ca. 1994** *An agent (A) is shown executing a job on a resource (R) via a gateway (G). Step 1: The agent and resource advertise themselves locally. Step 2: The gateway forwards the agent's unsatisfied request to Condor Pool B. Step 3: The matchmaker informs the two parties that they are potentially compatible. Step 4: The agent contacts the resource and executes a job via the gateway.*

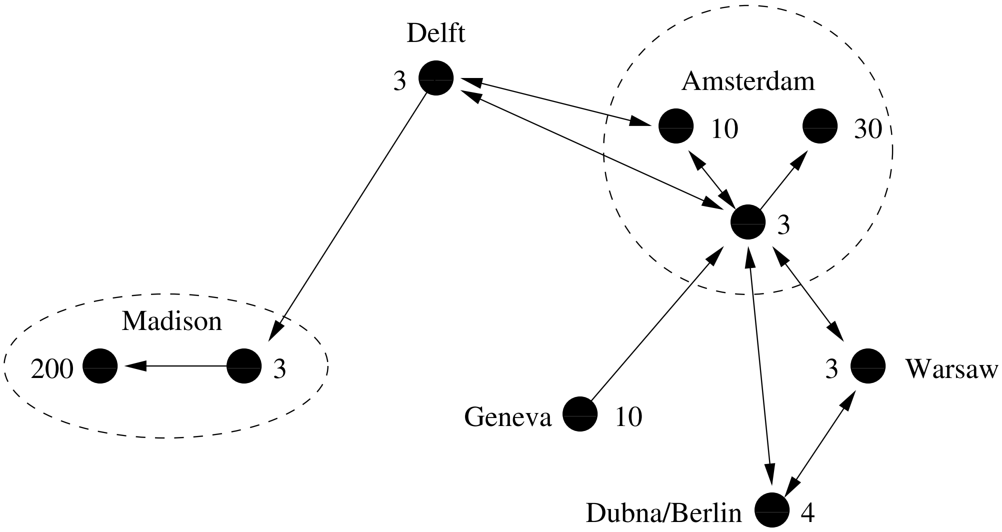


Figure 6. **Worldwide Condor Flock ca. 1994** *This is a map of the worldwide Condor flock in 1994. Each dot indicates a complete Condor pool. Numbers indicate the size of each Condor pool. Lines indicate flocking via gateways. Arrows indicate the direction that jobs may flow.*

enforced by established matchmakers. Even at the University of Wisconsin, researchers were unable to share resources between the separate engineering and computer science pools.

The concept of gateway flocking is shown in Figure 5. Here, the structure of two existing pools is preserved, while two gateway nodes pass information about participants between the

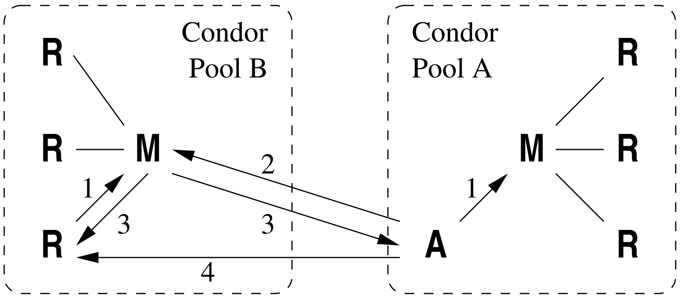


Figure 7. **Direct Flocking ca. 1998** *An agent (A) is shown executing a job on a resource (R) via direct flocking. Step 1: The agent and the resource advertise themselves locally. Step 2: The agent is unsatisfied, so it also advertises itself to Condor Pool B. Step 3: The matchmaker (M) informs the two parties that they are potentially compatible. Step 4: The agent contacts the resource and executes a job.*

two pools. If a gateway detects idle agents or resources in its home pool, it passes them to its peer, which advertises them in the remote pool, subject to the admission controls of the remote matchmaker. Gateway flocking is not necessarily bidirectional. A gateway may be configured with entirely different policies for advertising and accepting remote participants. Figure 6 shows the worldwide Condor flock in 1994.

The primary advantage of gateway flocking is that it is completely transparent to participants. If the owners of each pool agree on policies for sharing load, then cross-pool matches will be made without any modification by users. A very large system may be grown incrementally with administration only required between adjacent pools.

There are also significant limitations to gateway flocking. Because each pool is represented by a single gateway machine, the accounting of use by individual remote users is essentially impossible. Most importantly, gateway flocking only allows sharing at the organizational level — it does not permit an individual user to join multiple communities. This became a significant limitation as distributed computing became a larger and larger part of daily production work in scientific and commercial circles. Individual users might be members of multiple communities and yet not have the power or need to establish a formal relationship between both communities.

This problem was solved by *direct flocking*, shown in Figure 7. Here, an agent may simply report itself to multiple matchmakers. Jobs need not be assigned to any individual community, but may execute in either as resources become available. An agent may still use either community according to its policy while all participants maintain autonomy as before.

Both forms of flocking have their uses, and may even be applied at the same time. Gateway flocking requires agreement at the organizational level, but provides immediate and transparent benefit to all users. Direct flocking only requires agreement between one individual and another organization, but accordingly only benefits the user who takes the initiative.

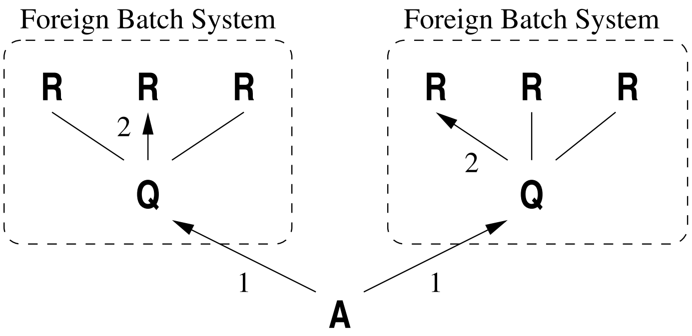


Figure 8. **Condor-G ca. 2000** *An agent (A) is shown executing two jobs through foreign batch queues (Q). Step 1: The agent transfers jobs directly to remote queues. Step 2: The jobs wait for idle resources (R), and then execute on them.*

This is a reasonable tradeoff found in everyday life. Consider an agreement between two airlines to cross-book each other's flights. This may require years of negotiation, pages of contracts, and complex compensation schemes to satisfy executives at a high level. But, once put in place, customer have immediate access to twice as many flights with no inconvenience. Conversely, an individual may take the initiative to seek service from two competing airlines individually. This places an additional burden on the customer to seek and use multiple services, but requires no herculean administrative agreement.

Although gateway flocking was of great use before the development of direct flocking, it did not survive the evolution of Condor. In addition to the necessary administrative complexity, it was also technically complex. The gateway participated in every interaction in the Condor kernel. It had to appear as both an agent and a resource, communicate with the matchmaker, and provide tunneling for the interaction between shadows and sandboxes. Any change to the protocol between any two components required a change to the gateway. Direct flocking, although less powerful, was much simpler to build and much easier for users to understand and deploy.

About 1998, a vision of a worldwide computational grid began to grow. [37] A significant early piece in the grid computing vision was a uniform interface for batch execution. The Globus Project [35] designed the Grid Resource Access and Management (GRAM) protocol [28] to fill this need. GRAM provides an abstraction for remote process queuing and execution with several powerful features such as strong security and file transfer. The Globus Project provides a server that speaks GRAM and converts its commands into a form understood by a variety of batch systems.

To take advantage of GRAM, a user still needs a system that can remember what jobs have been submitted, where they are, and what they are doing. If jobs should fail, the system must analyze the failure and re-submit the job if necessary. To track large numbers of jobs, users need queueing, prioritization, logging, and accounting. To provide this service, the Condor

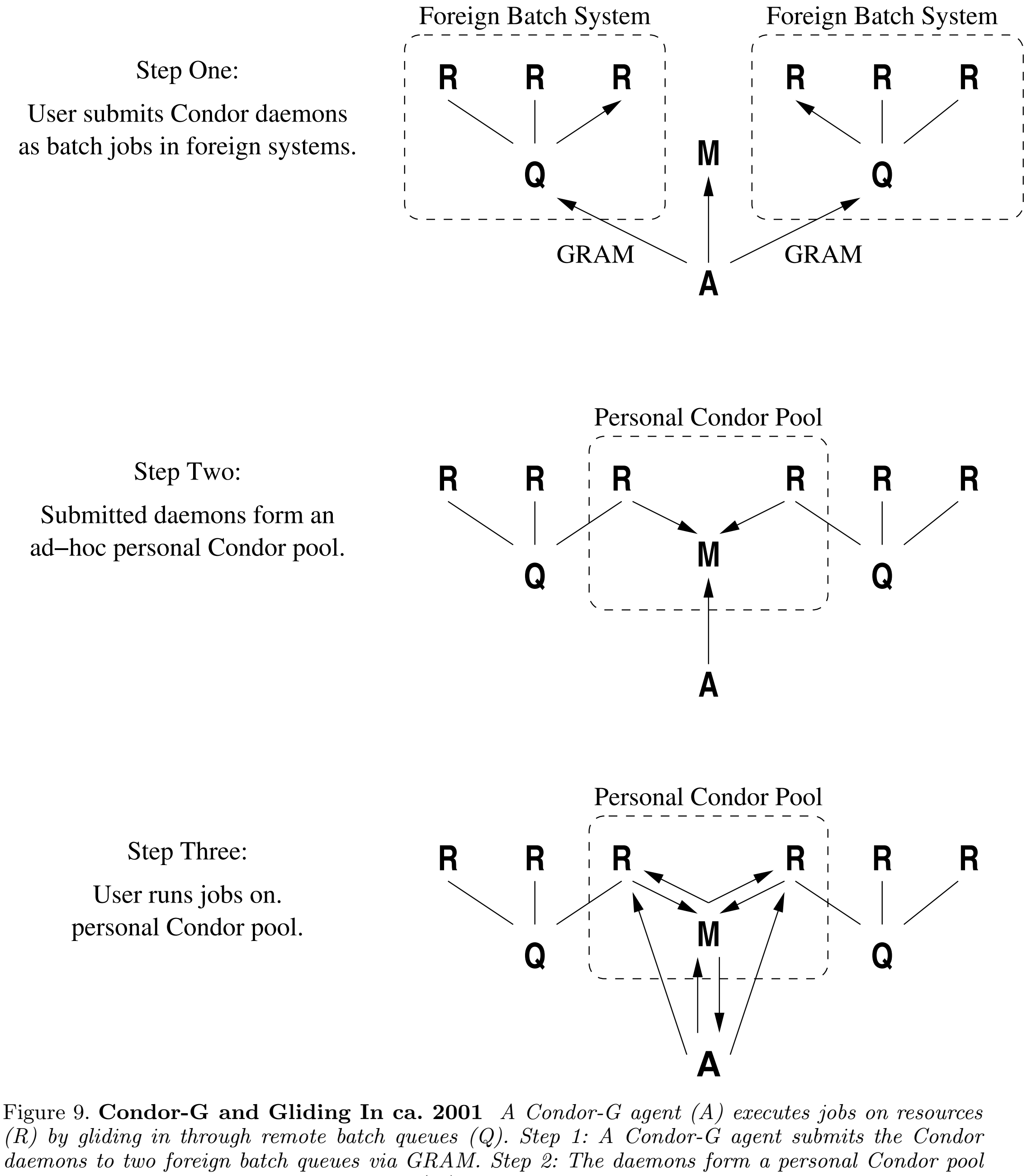


Figure 9. **Condor-G and Gliding In ca. 2001** *A Condor-G agent (A) executes jobs on resources (R) by gliding in through remote batch queues (Q). Step 1: A Condor-G agent submits the Condor daemons to two foreign batch queues via GRAM. Step 2: The daemons form a personal Condor pool with the user's personal matchmaker (M). Step 3: The agent executes jobs as in Figure 4.*

project adapted a standard Condor agent to speak GRAM, yielding a system called Condor-G, shown in Figure 8. This required some small changes to GRAM such as adding durability and two-phase commit to prevent the loss or repetition of jobs. [39]

The power of GRAM is to expand the reach of a user to any sort of batch system, whether it runs Condor or not. For example, the solution of the NUG30 [49] quadratic assignment problem relied on the ability of Condor-G to mediate access to over a thousand hosts spread across tens of batch systems on several continents. We will describe NUG30 in greater detail below.

The are also some disadvantages to GRAM. Primarily, it couples resource allocation and job execution. Unlike direct flocking in Figure 7, the agent must direct a particular job, with its executable image and all, to a particular queue without knowing the availability of resources behind that queue. This forces the agent to either over-subscribe itself by submitting jobs to multiple queues at once or under-subscribe itself by submitting jobs to potentially long queues. Another disadvantage is that Condor-G does not support all of the varied features of each batch system underlying GRAM. Of course, this is a necessity: if GRAM included all the bells and whistles of every underlying system, it would be so complex as to be unusable. However, a variety of useful features, such as the ability to checkpoint or extract the job's exit code are missing.

This problem is solved with a technique called *gliding in*, shown in Figure 9. To take advantage of both the powerful reach of Condor-G and the full Condor machinery, a personal Condor pool may be carved out of remote resources. This requires three steps. In the first step, a Condor-G agent is used to submit the standard Condor daemons as jobs to remote batch systems. From the remote system's perspective, the Condor daemons are ordinary jobs with no special privileges. In the second step, the daemons begin executing and contact a personal matchmaker started by the user. These remote resources along with the user's Condor-G agent and matchmaker from a personal Condor pool. In step three, the user may submit normal jobs to the Condor-G agent, which are then matched to and executed on remote resources with the full capabilities of Condor.

To this point, we have defined communities in terms of such concepts as responsibility, ownership, and control. However, communities may also be defined as a function of more tangible properties such as location, accessibility, and performance. Resources may group themselves together to express that they are "nearby" in measurable properties such as network latency or system throughput. We call these groupings *I/O communities.*

I/O communities were expressed in early computational grids such as the Distributed Batch Controller (DBC) [25]. The DBC was designed in 1996 for processing data from the NASA Goddard Space Flight Center. Two communities were included in the original design: one at the University of Wisconsin, and the other in the District of Columbia. A high level scheduler at Goddard would divide a set of data files among available communities. Each community was then responsible for transferring the input data, performing computation, and transferring the output back. Although the high-level scheduler directed the general progress of the computation, each community retained local control by employing Condor to manage its resources.

Another example of an I/O community is the *execution domain.* This concept was developed to improve the efficiency of data transfers across a wide-area network. An execution domain is a

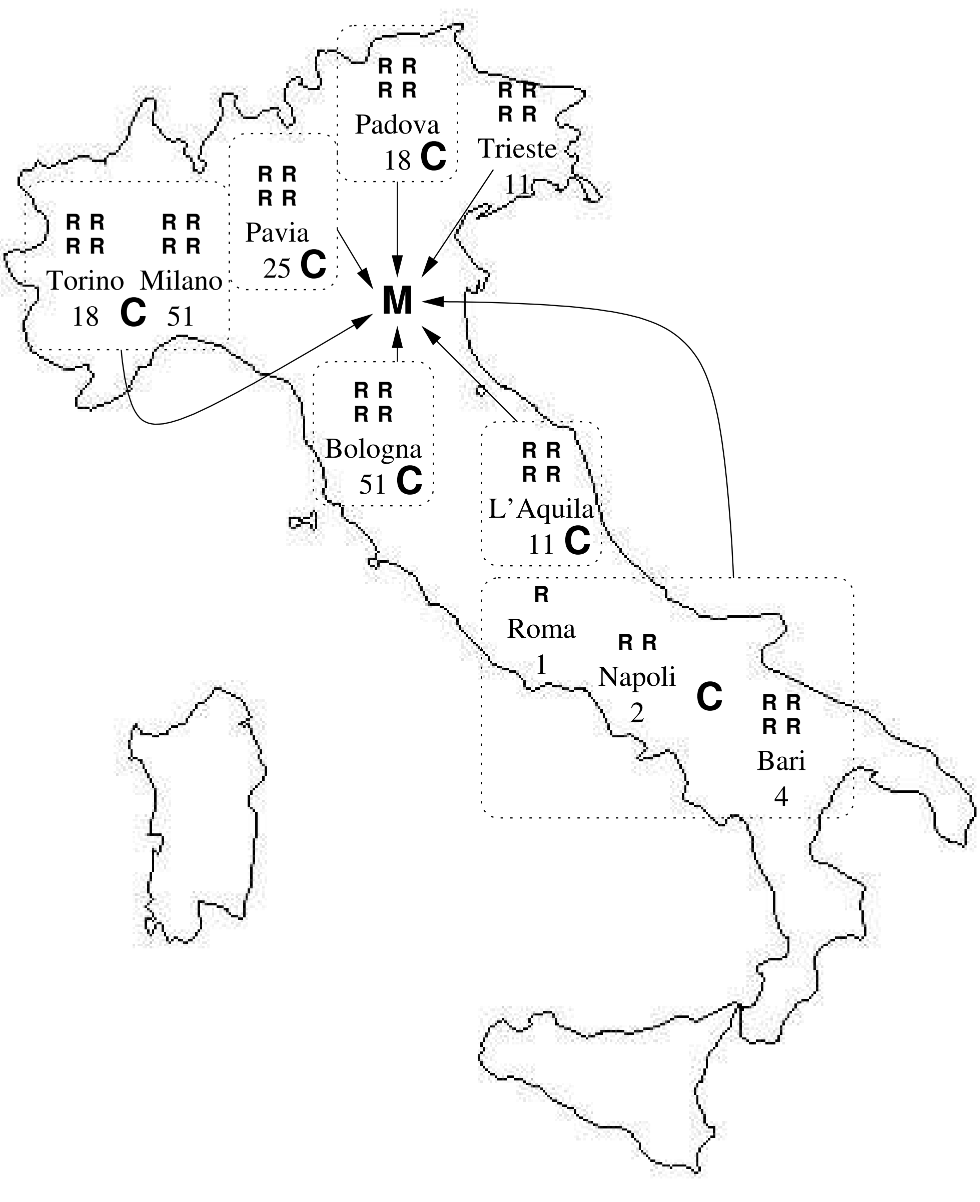


Figure 10. **INFN Condor Pool ca. 2002** *This is a map of a single Condor pool spread across Italy. All resources (R) across the country share the same matchmaker (M) in Bologna. Dotted lines indicate execution domains where resources share a checkpoint server (C). Numbers indicate resources at each site. Resources not assigned to a domain use the checkpoint server in Bologna.*

collection of resources that identify themselves with a checkpoint server that is close enough to provide good I/O performance. An agent may then make informed placement and migration decisions by taking into account the rough physical information provided by an execution domain. For example, an agent might strictly require that a job remain in the execution domain that it was submitted from. Or, it might permit a job to migrate out of its domain after a suitable waiting period. Examples of such policies expressed in the ClassAd language may be found in [18].

Figure 10 shows a deployed example of execution domains. The Istituto Nazionale de Fisica Nucleare (INFN) Condor pool consists of a large set of workstations spread across Italy. Although these resources are physically distributed, they are all part of a national organization, and thus share a common matchmaker in Bologna which enforces institutional policies. To encourage local access to data, six execution domains are defined within the pool, indicated by dotted lines. Each domain is internally connected by a fast network and shares a checkpoint server. Machines not specifically assigned to an execution domain default to the checkpoint server in Bologna.

Recently, the Condor project developed a complete framework for building general-purpose I/O communities. This framework permits access not only to checkpoint images, but also to executables and runtime data. This requires some additional machinery for all parties. The storage device must be an appliance with sophisticated naming and resource management. [19] The application must be outfitted with an interposition agent that can translate application I/O requests into the necessary remote operations. [75] Finally, an extension to the ClassAd language is necessary to expressing community relationships. This framework was used to improve the throughput of a high-energy physics simulation deployed on an international Condor flock. [74]

Planning and Scheduling

> *In preparing for battle I have always found that plans are useless, but planning is indispensable.*
>
> - Dwight D. Eisenhower (1890 - 1969)

The central purpose of distributed computing is to enable a community of users to perform work on a pool of shared resources. Because the number of jobs to be done nearly always outnumbers the available resources, somebody must decide how to allocate resources to jobs. Historically, this has been known as *scheduling.* A large amount of research in scheduling was motivated by the proliferation of massively parallel processor (MPP) machines in the early 1990s and the desire to use these very expensive resources as efficiently as possible. Many of the resource management systems we have mentioned contain powerful scheduling components in their architecture.

Yet, grid computing cannot be served by a centralized scheduling algorithm. By definition, a grid has multiple owners. Two supercomputers purchased by separate organizations with distinct funds will never share a single scheduling algorithm. The owners of these resources will rightfully retain ultimate control over their own machines and may change scheduling policies according to local decisions. Therefore, we draw a distinction based on the ownership. Grid computing requires both *planning* and *scheduling*

Planning is the acquisition of resources by users. Users are typically interested in increasing personal metrics such as response time, turnaround time, and throughput of their own jobs within reasonable costs. For example, an airline customer performs planning when she examines all available flights from Madison to Melbourne in an attempt to arrive before Friday for less than $1500. Planning is usually concerned with the matters of *what* and *where*.

Scheduling is the management of a resource by its owner. Resource owners are typically interested in increasing system metrics such as efficiency, utilization, and throughput without losing the customers they intend to serve. For example, an airline performs scheduling when its sets the routes and times that its planes travel. It has an interest in keeping planes full and prices high without losing customers to its competitors. Scheduling is usually concerned with the matters of *who* and *when*.

Of course, there is feedback between planning and scheduling. Customers change their plans when they discover a scheduled flight is frequently late. Airlines change their schedules according to the number of customers that actually purchase tickets and board the plane. But both parties retain their independence. A customer may purchase more tickets than she actually uses. An airline may change its schedules knowing full well it will lose some customers. Each side must weigh the social and financial consequences against the benefits.

The challenges faced by planning and scheduling in a grid computing environment are very similar to the challenges faced by cycle-scavenging from desktop workstations. The insistence that each desktop workstation is the sole property of one individual who is in complete control, characterized by the success of the personal computer, results in distributed ownership. Personal preferences and the fact that desktop workstations are often purchased, upgraded, and configured in a haphazard manner results in heterogeneous resources. Workstation owners powering their machines on and off whenever they desire creates a dynamic resource pool, and owners performing interactive work on their own machines creates external influences.

Condor uses *matchmaking* to bridge the gap between planning and scheduling. Matchmaking creates opportunities for planners and schedulers to work together while still respecting their essential independence. Although Condor has traditionally focused on producing robust planners rather than complex schedulers, the matchmaking framework allows both parties to implement sophisticated algorithms.

Matchmaking requires four steps, shown in Figure 11. In the first step, agents and resources advertise their characteristics and requirements in *classified advertisements* (ClassAds), named after brief advertisements for goods and services found in the morning newspaper. In the second step, a *matchmaker* scans the known ClassAds and creates pairs that satisfy each other's constraints and preferences. In the third step, the matchmaker informs both parties of the match. The responsibility of the matchmaker then ceases with respect to the match. In the final step, *claiming*, the matched agent and resource establish contact, possibly negotiate further terms, and then cooperate to execute a job. The clean separation of the *claiming* step has noteworthy advantages, such as enabling the resource to independently authenticate and authorize the match, and enabling the resource to verify that match constraints are still satisfied with respect to current conditions. [54]

Prepared using cpeauth.cls

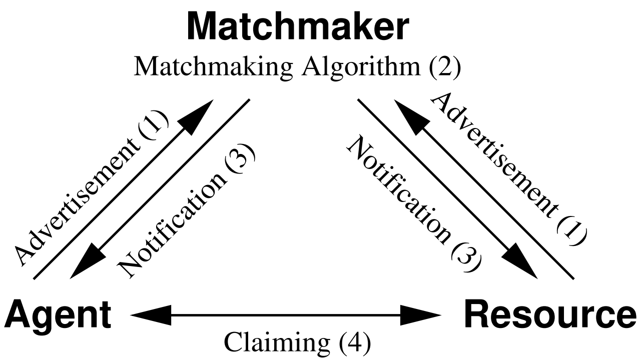


Figure 11. **Matchmaking**

Job ClassAd	Machine ClassAd
[	[
MyType = “Job”	**MyType** = “Machine”
TargetType = “Machine”	**TargetType** = “Job”
Requirements =	**Machine** = “nostos.cs.wisc.edu”
((other.Arch==“INTEL” &&	**Requirements** =
other.OpSys==“LINUX”)	(LoadAvg <= 0.300000) &&
&& other.Disk > my.DiskUsage)	(KeyboardIdle > (15 * 60))
Rank = (Memory * 10000) + KFlops	**Rank** = other.Department==self.Department
Cmd = “/home/tannenba/bin/sim-exe”	**Arch** = “INTEL”
Department = “CompSci”	**OpSys** = “LINUX”
Owner = “tannenba”	**Disk** = 3076076
DiskUsage = 6000	]
]	

Figure 12. **Two Sample ClassAds from Condor.**

A ClassAd is a set of uniquely named expressions, using a semi-structured data model so no specific schema is required by the matchmaker. Each named expression is called an *attribute*. Each attribute has an *attribute name* and an *attribute value*. In our initial ClassAd implementation, the attribute value could be a simple integer, string, floating point value, or expression comprised of arithmetic and logical operators. After gaining more experience, we created a second ClassAd implementation which introduced richer attribute value types and related operators for records, sets, and tertiary conditional operators similar to C.

Because ClassAds are schema-free, participants in the system may attempt to refer to attributes that do not exist. For example, an job may prefer machines with the attribute `(Owner == ‘‘Fred’’)`, yet some machines may fail to define the attribute `Owner`. To solve this, ClassAds use *three-valued logic* which allows expressions to evaluated to either `true`, `false`, or `undefined`. This explicit support for missing information allows users to build robust requirements even without a fixed schema.

The Condor matchmaker assigns significance to two special attributes: `Requirements` and `Rank`. `Requirements` indicates a constraint and `Rank` measures the desirability of a match. The matchmaking algorithm requires that for two ClassAds to match, both of their corresponding `Requirements` must evaluate to `true`. The `Rank` attribute should evaluate to an arbitrary floating point number. `Rank` is used to choose among compatible matches: Among provider ads matching a given customer ad, the matchmaker chooses the one with the highest Rank value (noninteger values are treated as zero), breaking ties according to the provider's Rank value.

ClassAds for a job and a machine are shown in Figure 12. The `Requirements` state that the job must be matched with an Intel Linux machine which has enough free disk space (more than 6 megabytes). Out of any machines which meet these requirements, the job prefers a machine with lots of memory, followed by good floating point performance. Meanwhile, the machine ad `Requirements` states that this machine is not willing to match with any job unless its load average is low and the keyboard has been idle for more than 15 minutes. In other words, it is only willing to run jobs when it would otherwise sit idle. When it is willing to run a job, the `Rank` expression states it prefers to run jobs submitted by users from its own department.

Combinations of Planning and Scheduling

As we mentioned above, planning and scheduling are related yet independent. Both planning and scheduling can be combined within on system.

Condor-G, for instance, can perform *planning around a schedule*. Remote site schedulers control the resources, and once Condor-G submits a job into a remote queue, when it will actually run is at the mercy of the remote scheduler (see Figure 8). But if the remote scheduler publishes information about its timetable or workload priorities via a ClassAd to the Condor-G matchmaker, Condor-G could begin making better choices by planning where it should submit jobs (if authorized at multiple sites), when it should submit them, and/or what types of jobs to submit. In fact, this approach is currently being investigated by the PPDG. [10] As more information is published, Condor-G can perform better planning. But even in a complete absence of information from the remote scheduler, Condor-G could still perform planning, although the plan may start to resemble "shooting in the dark". For example, one such plan could be to submit the job once to each site willing to take it, wait and see where it completes first, and then upon completion, delete the job from the remaining sites.

Another combination is *scheduling within a plan*. Consider as an analogy a large company which purchases, in advance, eight seats on a Greyhound bus each week for a year. The company does not control the bus schedule, so they must plan how to utilize the buses. However, after purchasing the tickets, the company is free to decide to send to the bus terminal whatever employees it wants in whatever order it desires. The Condor system performs scheduling within a plan in several situations. One such situation is when Condor schedules parallel jobs on compute clusters. [79] When the matchmaking framework offers a match to an agent and the subsequent claiming protocol is successful, the agent considers itself the owner of that resource

Prepared using cpeauth.cls

until told otherwise. The agent then creates a schedule for running tasks upon the resources which it has claimed via planning.

Matchmaking in Practice

Matchmaking emerged over several versions of the Condor software. The initial system used a fixed structure for representing both resources and jobs. As the needs of the users developed, these structures went through three major revisions, each introducing more complexity in an attempt to retain backwards compatibility with the old. This finally led to the realization that no fixed schema would serve for all time and resulted in the development of a C-like language known as *control expressions* [22] in 1992. By 1995, the expressions had been generalized into *classified advertisements* or ClassAds. [58] This first implementation is still used heavily in Condor at the time of this writing. However, it is slowly being replaced by a new implementation [62, 63, 61] which incorporated lessons from language theory and database systems.

A standalone open source software package for manipulating ClassAds is available in both Java and C++. [72] This package enables the matchmaking framework to be used in other distributed computing projects. [77, 32] Several research extensions to matchmaking have been built. *Gang matching* [63, 61] permits the co-allocation of more than once resource, such as a license and a machine. *Collections* provide persistent storage for large numbers of ClassAds with database features such as transactions and indexing. *Set matching* [15] permits the selection and claiming of large numbers of resource using a very compact expression representation. *Indirect references* [74] permit one ClassAd to refer to another and facilitate the construction of the I/O communities mentioned above.

In practice, we have found matchmaking with ClassAds to be very powerful. Most resource management systems allow customers to set provide requirements and preferences on the resources they wish. But the matchmaking framework's ability to allow resources to impose constraints on the customers they wish to service is unique and necessary for preserving distributed ownership. The clean separation between matchmaking and claiming allows the matchmaker to be blissfully ignorant about the actual mechanics of allocation, permitting it to be a general service which does not have to change when new types of resources or customers are added. Because stale information may lead to a bad match, a resource is free to refuse a claim even after it has been matched. Matchmaking is capable of representing wildly divergent resources, ranging from electron microscopes to storage arrays because resources are free to describe themselves without a schema. Even with similar resources, organizations track different data, so no schema promulgated by the Condor software would be sufficient. Finally, the matchmaker is stateless and thus can scale to very large systems without complex failure recovery.

Problem Solvers

We have delved down into the details of planning and execution that the user relies upon, but may never see. Let us now move up in the Condor kernel and discuss the environment in which a user actually works.

A *problem solver* is a higher-level structure built on top of the Condor agent. Two problem solvers are provided with Condor: *master-worker* and the *directed acyclic graph manager*. Each provides a unique programming model for managing large numbers of jobs. Other problem solvers are possible and may be built using the public interfaces of a Condor agent.

A problem solver relies on a Condor agent in two important ways. A problem solver uses the agent as a service for reliably executing jobs. It need not worry about the many ways that a job may fail in a distributed system, because the agent assumes all responsibility for hiding and retrying such errors. Thus, a problem solver need only concern itself with the application-specific details of ordering and task selection. The agent is also responsible for making the problem solver itself reliable. To accomplish this, the problem solver is presented as a normal Condor job which simply executes at the submission site. Once started, the problem solver may then turn around and submit sub-jobs back to the agent.

From the perspective of a user or a problem solver, a Condor agent is identical to a Condor-G agent. Thus, any of the structures we describe below may be applied to an ordinary Condor pool or to a wide-area grid computing scenario.

Master-Worker

Master-Worker (MW) is a system for solving a problem of indeterminate size on a large and unreliable workforce. The MW model is well-suited for problems such as parameter searches where large portions of the problem space may be examined independently, yet the progress of the program is guided by intermediate results.

The MW model is shown in Figure 13. One master process directs the computation with the assistance of as many remote workers as the computing environment can provide. The master itself contains three components: a *work list*, a *tracking* module, and a *steering* module. The work list is simply a record of all outstanding work the master wishes to be done. The tracking module accounts for remote worker processes and assigns them uncompleted work. The steering module directs the computation by examining results, modifying the work list, and communicating with Condor to obtain a sufficient number of worker processes.

Of course, workers are inherently unreliable: they disappear when machines crash and they reappear as new resources become available. If a worker should disappear while holding a work unit, the tracking module simply returns it to the work list. The tracking module may even take additional steps to replicate or reassign work for greater reliability or simply to speed the completion of the last remaining work units.

MW is packaged as source code for several C++ classes. The user must extend the classes to perform the necessary application-specific worker processing and master assignment, but all of the necessary communication details are transparent to the user.

MW is the result of several generations of software development. It began with J. Pruyne's doctoral thesis [58], which proposed that applications ought to have an explicit interface to

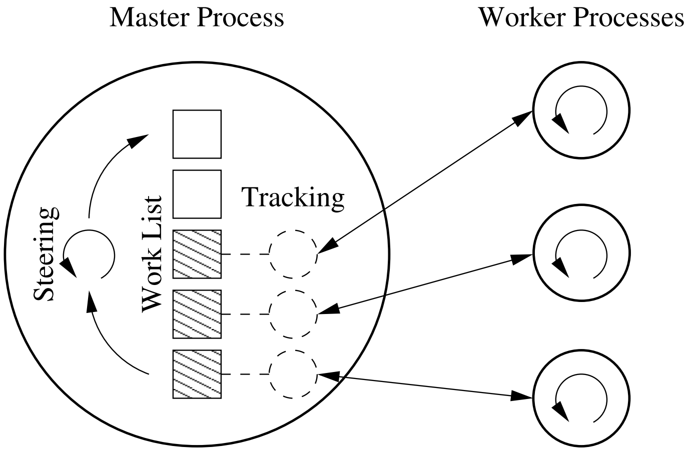


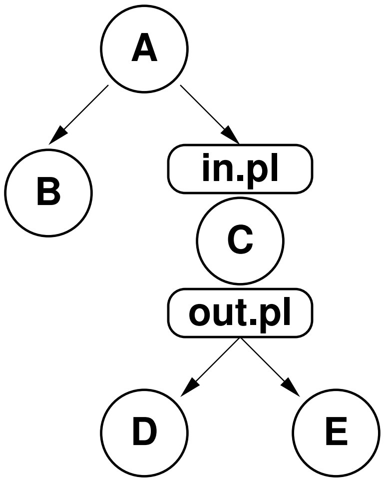
Figure 13. **Structure of a Master-Worker Program**

the system responsible for finding resources and placing jobs. Such changes were contributed to PVM release 3.3. [60] The first user of this interface was the Worker Distributor (WoDi or "Woody"), which provided a simple interface to a work list processed by a large number of workers. The WoDi interface was a very high-level abstraction that presented no fundamental dependencies on PVM. It was quickly realized that the same functionality could be built entirely without PVM. Thus, MW was born. [49] MW provides an interface similar to WoDi, but has several interchangeable implementations. Today, MW can operate by communicating through PVM, through a shared file system, over sockets, or using the standard universe (described below).

Directed Acyclic Graph Manager

The Directed Acyclic Graph Manager (DAGMan) is a service for executing multiple jobs with dependencies in a declarative form. DAGMan might be thought of as a distributed, fault-tolerant version of the traditional `make`. Like its ancestor, it accepts a declaration that lists the work to be done and the constraints on its order. Unlike `make`, it does not depend on the file system to record a DAG's progress. Indications of completion may be scattered across a distributed system, so DAGMan keeps private logs, allowing it to resume a DAG where it left off, even in the face of crashes and other failures.

Figure 14 demonstrates the language accepted by DAGMan. A `JOB` statement associates an abstract name (`A`) with a file (`a.condor`) that describes a complete Condor job. A `PARENT-CHILD` statement describes the relationship between two or more jobs. In this script, jobs `B` and `C` are may not run until `A` has completed, while jobs `D` and `E` may not run until `C`




```
JOB A a.condor
JOB B b.condor
JOB C c.condor
JOB D d.condor
JOB E e.condor
PARENT A CHILD B C
PARENT C CHILD D E
SCRIPT PRE C in.pl
SCRIPT POST C out.pl
RETRY C 3
```

Figure 14. **A Directed Acyclic Graph**

has completed. Jobs that are independent of each other may run in any order and possibly simultaneously.

In this script, job `C` is associated with a `PRE` and a `POST` program. These commands indicate programs to be run before and after a job executes. `PRE` and `POST` programs are not submitted as Condor jobs, but are run by DAGMan on the submitting machine. `PRE` programs are generally used to prepare the execution environment by transferring or uncompressing files, while `POST` programs are generally used to tear down the environment or to evaluate the output of the job.

DAGMan presents an excellent opportunity to study the problem of multi-level error processing. In a complex system that ranges from the high-level view of DAGs all the way down to the minutiae of remote procedure calls, it is essential to tease out the source of an error to avoid unnecessarily burdening the user with error messages.

Jobs may fail because of the nature of the distributed system. Network outages and reclaimed resources may cause Condor to lose contact with a running job. Such failures are not indications that the job itself has failed, but rather that the system has failed. Such situations are detected and retried by the agent in its responsibility to execute jobs reliably. DAGMan is never aware of such failures.

Jobs may also fail of their own accord. A job may produce an ordinary error result if the user forgets to provide a necessary argument or input file. In this case, DAGMan is aware that the job has completed and sees a program result indicating an error. It responds by writing out a rescue DAG and exiting with an error code. The *rescue DAG* is a new DAG listing the elements of the original DAG left unexecuted. To remedy the situation, the user may examine the rescue DAG, fix any mistakes in submission, and resubmit it as a normal DAG.

Prepared using cpeauth.cls

Some environmental errors go undetected by the distributed system. For example, a corrupted executable or a dismounted file system *should* be detected by the distributed system and retried at the level of the agent. However, if the job was executed via Condor-G through a foreign batch system, such detail beyond "job failed" may not be available, and the job will appear to have failed of its own accord. For these reasons, DAGMan allows the user to specify that a failed job be retried, using the `RETRY` command shown in Figure 14.

Some errors may be reported in unusual ways. Some applications, upon detecting a corrupt environment, do not set an appropriate exit code, but simply produce a message on the output stream and exit with an indication of success. To remedy this, the user may provide a `POST` script that examines the program's output for a valid format. If not found, the `POST` script may return failure, indicating that the job has failed and triggering a `RETRY` or the production of a rescue DAG.

Split Execution

So far, this chapter has explored many of the techniques of getting a job to an appropriate execution site. However, that only solves part of the problem. Once placed, a job may find itself in a hostile environment: it may be without the files it needs, it may be behind a firewall, or it may not even have the necessary user credentials to access its data. Worse yet, few resources sites are uniform in their hostility. One site may have a user's files yet not recognize the user, while another site may have just the opposite situation.

No single party can solve this problem. No process has all the information and tools necessary to reproduce the user's home environment. Only the execution machine knows what file systems, networks, and databases may be accessed and how they must be reached. Only the submission machine knows at runtime what precise resources the job must actually be directed to. Nobody knows in advance what names the job may find its resources under, as this is a function of location, time, and user preference.

Cooperation is needed. We call this cooperation *split execution*. It is accomplished by two distinct components: the *shadow* and the *sandbox*. These were mentioned in Figure 3 above. Here we will examine them in detail.

The *shadow* represents the user to the system. It is responsible for deciding exactly what the job must do as it runs. The shadow provides absolutely everything needed to specify the job at runtime: the executable, the arguments, the environment, the input files, and so on. None of this is made known outside of the agent until the actual moment of execution. This allows the agent to defer placement decisions until the last possible moment. If the agent submits requests for resources to several matchmakers, it may award the highest priority job to the first resource that becomes available without breaking any previous commitments.

The *sandbox* is responsible for giving the job a safe place to play. It must ask the shadow for the job's details and then create an appropriate environment. The sandbox really has two distinct components: the *sand* and the *box*. The sand must make the job feel at home by providing everything that it needs to run correctly. The box must protect the resource from any harm that a malicious job might cause. The box has already received much attention [47, 64, 20, 65], so we will focus here on describing the sand. *

Condor provides several *universes* that create a specific job environment. A universe is defined by a matched sandbox and shadow, so the development of a new universe necessarily requires the deployment of new software modules at both sides. The matchmaking framework described above can be used to select resources equipped with the appropriate universe. Here, we will describe the oldest and the newest universes in Condor: the standard universe and the Java universe.

The Standard Universe

The standard universe was the only universe supplied by the earliest versions of Condor and is a descendant of the Remote UNIX [52] facility.

The goal of the standard universe is to faithfully reproduce the user's home POSIX environment for a single process running at a remote site. The standard universe provides emulation for the vast majority of standard system calls including file I/O, signal routing, and resource management. Process creation and inter-process communication are not supported and users requiring such features are advised to consider the MPI and PVM universes or the MW problem solver, all described above.

The standard universe also provides *checkpointing*. This is the ability to take a snapshot of a running process and place it in stable storage. The snapshot may then be moved to another site and the entire process reconstructed and then resumed right from where it left off. This may be done to migrate a process from one machine to another, or it may be used to recover failed processes and improve throughput in the face of failures.

Figure 15 shows all of the components necessary to create the standard universe. At the execution site, the sandbox is responsible for creating a safe and usable execution environment. It prepares the machine by creating a temporary directory for the job, and then fetches all of the job's details — the executable, environment, arguments, and so on — and places them in the execute directory. It then invokes the job and is responsible for monitoring its health, protecting it from interference, and destroying it if necessary.

At the submission site, the shadow is responsible for representing the user. It provides all of the job details for the sandbox and makes all of the necessary policy decisions about the job as it runs. In addition, it provides an I/O service accessible over a secure RPC channel. This provides remote access to the user's home storage device.

To communicate with the shadow, the user's job must be re-linked with a special library provided by Condor. This library has the same interface as the standard C library, so no changes to the user's code are necessary. The library converts all of the job's standard system calls into secure remote procedure calls back to the shadow. It is also capable of converting I/O operations into a variety of remote access protocols, including HTTP, GridFTP [13], NeST

*The Paradyn Project has explored several variations of this problem, such as attacking the sandbox [55], defending the shadow [40], and hijacking the job. [80]

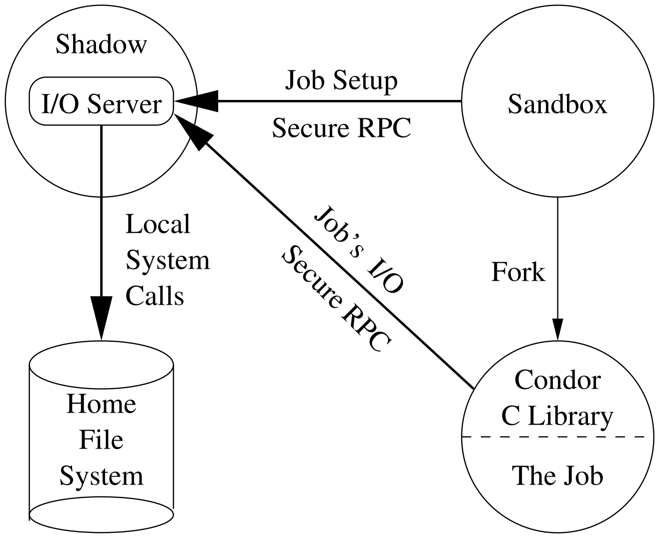


Figure 15. **The Standard Universe**

[19], and Kangaroo [73]. In addition, it may apply a number of other transformations, such as buffering, compression, and speculative I/O.

It is vital to note that the shadow remains in control of the entire operation. Although both the sandbox and the Condor library are equipped with powerful mechanisms, neither is authorized to make decisions without the shadow's consent. This maximizes the flexibility of the user to make run-time decisions about exactly what runs where and when.

An example of this principle is the *two-phase open*. Neither the sandbox nor the library is permitted to simply open a file by name. Instead, they must first issue a request to map a logical file name (the application's argument to `open`) into a physical file name. The physical file name is similar to a URL and describes the actual file name to be used, the method by which to access it, and any transformations to be applied.

Figure 16 demonstrates two-phase open. Here the application requests a file named `alpha`. The library asks the shadow how the file should be accessed. The shadow responds that the file is available using remote procedure calls, but is compressed and under a different name. the library then issues an `open` to access the file.

Another example is given in Figure 17. Here the application requests a file named `beta`. The library asks the shadow how the file should be accessed. The shadow responds that the file is available using the NeST protocol on a server named `nest.wisc.edu`. The library then contacts that server and indicates success to the user's job.

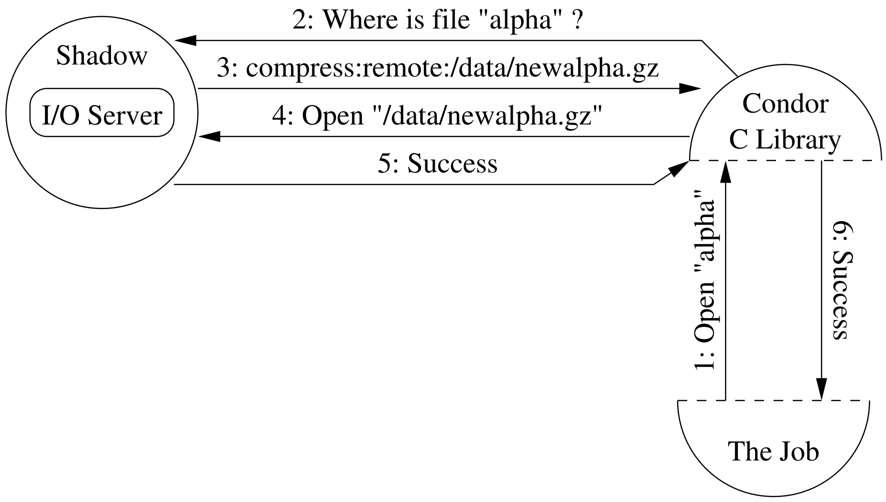


Figure 16. **Two-Phase Open**

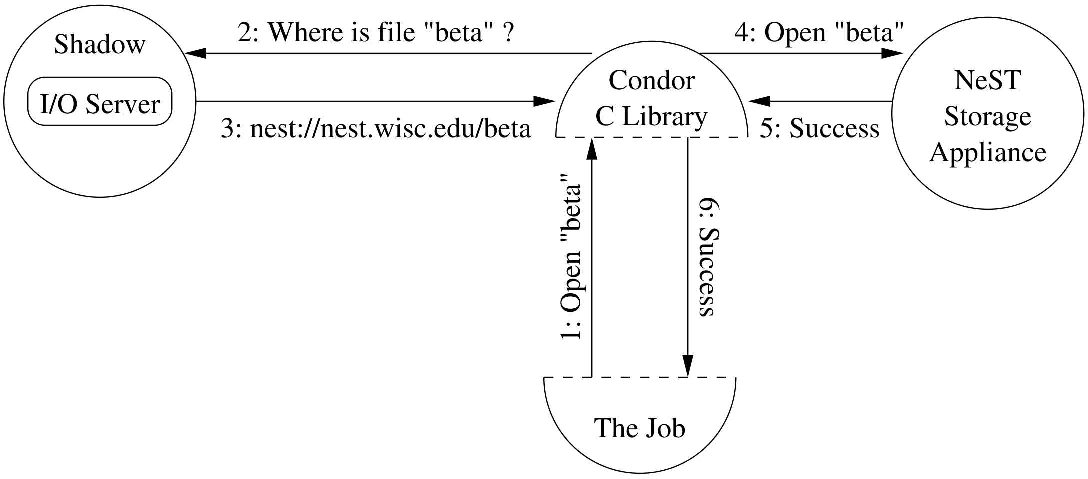


Figure 17. **Two-Phase Open**

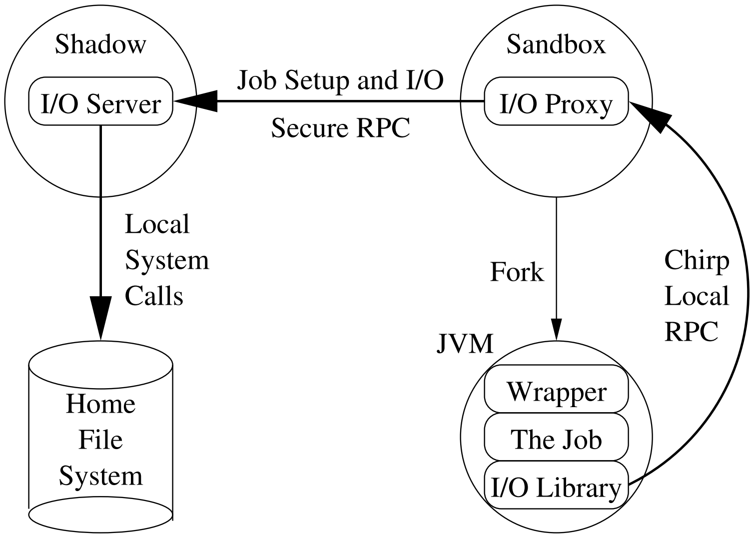


Figure 18. **The Java Universe**

The mechanics of checkpointing and remote system calls in Condor are described in great detail by Litzkow et al [66, 51]. We have also described Bypass, a standalone system for building similar split execution systems outside of Condor [75].

The Java Universe

A universe for Java programs was added to Condor in late 2001. This was due to a growing community of scientific users that wished to perform simulations and other work in Java. Although such programs might run slower than native code, such losses were offset by faster development times and access to larger numbers of machines. By targeting applications to the Java Virtual Machine (JVM), users could avoid dealing with the time-consuming details of specific computing systems.

Previously, users had run Java programs in Condor by submitting an entire JVM binary as a standard universe job. Although this worked, it was inefficient on two counts: the JVM binary could only run on one type of CPU, which defied the whole point of a universal instruction set; and the repeated transfer of the JVM and the standard libraries was a waste of resources on static data.

A new Java universe was developed which would raise the level of abstraction to create a complete Java environment rather than a POSIX environment. The components of the new Java universe are shown in Figure 18. The responsibilities of each component are the same as other universes, but the functionality changes to accommodate the unique features of Java.

The sandbox is responsible for creating a safe and comfortable execution environment. It must ask the shadow for all of the job's details, just as in the standard universe. However, the location of the JVM is provided by the local administrator, as this may change from machine to machine. In addition, a Java program consists of a variety of runtime components, including class files, archive files, and standard libraries. The sandbox must place all of these components in a private execution directory along with the user's credentials and start the JVM according to the local details.

The I/O mechanism is somewhat more complicated in the Java universe. The job is linked against a Java I/O library that presents remote I/O in terms of standard interfaces such as `InputStream` and `OutputStream`. This library does not communicate directly with any storage device, but instead calls an I/O proxy managed by the sandbox. This unencrypted connection is secure by making use of the loopback network interface and presenting a shared secret. The sandbox then executes the job's I/O requests along the secure RPC channel to the shadow, using all of the same security mechanisms and techniques as in the standard universe.

Initially, we chose this I/O mechanism so as to avoid re-implementing all of the I/O and security features in Java and suffering the attendant maintenance work. However, there are several advantages of the I/O proxy over the more direct route used by the standard universe. The proxy allows the sandbox to pass through obstacles that the job does not know about. For example, if a firewall lies between the execution site and the job's storage, the sandbox may use its knowledge of the firewall to authenticate and pass through. Likewise, the user may provide credentials for the sandbox to use on behalf of the job without rewriting the job to make use of them.

The Java universe is sensitive to a wider variety of errors than most distributed computing environments. In addition to all of the usual failures that plague remote execution, the Java environment is notoriously sensitive to installation problems, and many jobs and sites are unable to find runtime components, whether they are shared libraries, Java classes, or the JVM itself. Unfortunately, many of these environmental errors are presented to the job itself as ordinary exceptions, rather than expressed to the sandbox as an environmental failure. To combat this problem, a small Java wrapper program is used to execute the user's job indirectly and analyze the meaning of any errors in the execution. A complete discussion of this problem and its solution may be found in [76].

Case Studies

Grid technology, and Condor in particular, is working today on real-world problems. The three brief case studies presented below provide a glimpse on how Condor and/or Condor-G is being used in production not only in academia, but also in industry. Two commercial organizations, with the foresight to embrace the integration of computational grids into their operations, are presented.

Commercial Case Studies

Micron Technology, Inc.

Micron Technology, Inc., has established itself as one of the leading worldwide providers of semiconductor solutions. Micron's quality semiconductor solutions serve customers in a variety of industries including computer and computer-peripheral manufacturing, consumer electronics, CAD/CAM, telecommunications, office automation, networking and data processing, and graphics display.

Micron's mission is to be the most efficient and innovative global provider of semiconductor solutions. This mission is exemplified by short cycle times, high yields, low production costs, and die sizes that are some of the smallest in the industry. To meet these goals, manufacturing and engineering processes are tightly controlled at all steps, requiring significant computational analysis.

Before Condor, Micron had to purchase dedicated compute resources to meet peak demand for engineering analysis tasks. Condor's ability to consolidate idle compute resources across the enterprise offered Micron the opportunity to meet its engineering needs without incurring the cost associated with traditional, dedicated compute resources. With over 18,000 employees worldwide, Micron was enticed by the thought of unlocking the computing potential of its desktop resources.

So far, Micron has setup two primary Condor pools that contain a mixture of desktop machines and dedicated compute servers. Condor manages the processing of tens of thousands of engineering analysis jobs per week. Micron engineers report that the analysis jobs run faster and require less maintenance. As an added bonus, dedicated resources that were formerly used for both compute intensive analysis and less intensive reporting tasks can now be used solely for compute intensive processes with greater efficiency.

Advocates of Condor at Micron especially like how easy it has been to deploy Condor across departments, due to the clear model of resource ownership and sandboxed environment. Micron's software developers, however, would like to see better integration of Condor with a wider variety of middleware solutions, such as messaging or CORBA.

C.O.R.E. Digital Pictures Inc.

C.O.R.E. Digital Pictures is a highly successful Toronto-based computer animation studio, co-founded in 1994 by William Shatner (of film and television fame) and four talented animators.

Photo-realistic animation, especially for cutting-edge film special effects, is a compute intensive process. Each frame can take up to an hour, and one second of animation can require 30 or more frames. When the studio was first starting out and had only a dozen employees, each animator would handle their own render jobs and resources by hand. But with lots of rapid growth and the arrival of multiple major motion picture contracts, it became evident that this approach would no longer be sufficient. In 1998, C.O.R.E. looked into several RMS packages and settled upon Condor.

Today, Condor manages a pool consisting of 70 Linux machines and 21 Silicon Graphics machines. The 70 Linux machines are all dual-CPU and mostly reside on the desktops of the

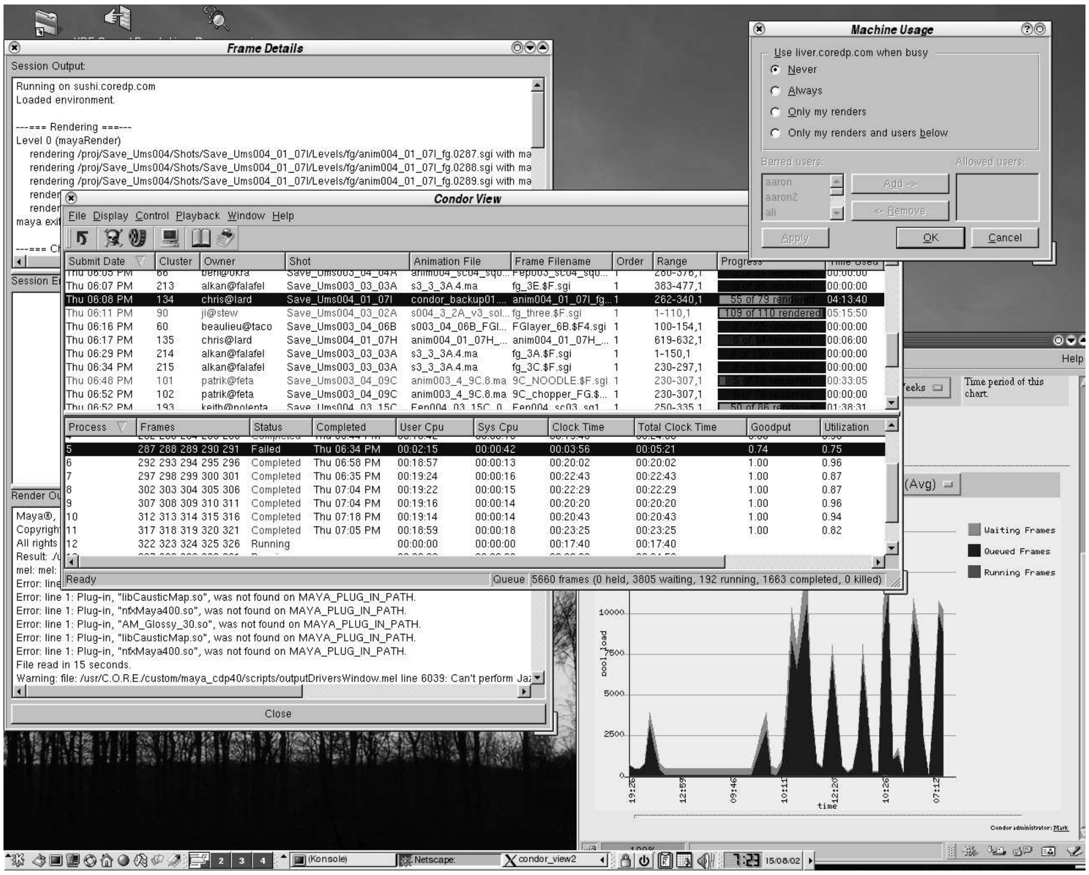


Figure 19. **Vertical Integration of Condor for Computer Animation.** *Custom GUI and database integration tools sitting on top of Condor help computer animators at C.O.R.E Digital Pictures.*

animators. By taking advantage of Condor ClassAds and native support for multi-processor machines, one CPU is dedicated to running Condor jobs while the second CPU only runs jobs when the machine is not being used interactively by its owner.

Each animator has his own Condor queuing agent on his own desktop. On a busy day, C.O.R.E. animators submit over 15,000 jobs to Condor. C.O.R.E. has done a significant amount of vertical integration to fit Condor transparently into their daily operations. Each animator interfaces with Condor via a set of custom tools tailored to present Condor's operations in terms of a more familiar animation environment (see Figure 19). C.O.R.E. developers created a session meta-scheduler that interfaces with Condor in a manner similar to the DAGMan service previously described. When an animator hits the "render" button, a new session is

Table I. **NUG30 Computation Statistics.** *Part A lists how many CPUs were utilized at different locations on the grid during the seven day NUG30 run. Part B lists other interesting statistics about the run.*

Part A

Number	Architecture	Location
1024	SGI/Irix	NCSA
414	Intel/Linux	Argonne
246	Intel/Linux	U. of Wisconsin
190	Intel/Linux	Georgia Tech
146	Intel/Solaris	U. of Wisconsin
133	Sun/Solaris	U. of Wisconsin
96	SGI/Irix	Argonne
94	Intel/Solaris	Georgia Tech
54	Intel/Linux	Italy (INFN)
45	SGI/Irix	NCSA
25	Intel/Linux	U. of New Mexico
16	Intel/Linux	NCSA
12	Sun/Solaris	Northwestern U.
10	Sun/Solaris	Columbia U.
5	Intel/Linux	Columbia U.

Part B

Total number of CPUs utilized	2510
Average number of simultaneous CPUs	652.7
Maximum number of simultaneous CPUs	1009
Running wall clock time (sec)	597,872
Total CPU time consumed (sec)	346,640,860
Number of times a machine joined the computation	19,063
Equivalent CPU time (sec) on an HP C3000 workstation	218,823,577

created and the custom meta-scheduler is submitted as a job into Condor. The meta-scheduler translates this session into a series of rendering jobs which it subsequently submits to Condor, asking Condor for notification on their progress. As Condor notification events arrive, this triggers the meta-scheduler to update a database and perhaps submit follow-up jobs following a DAG.

C.O.R.E. makes considerable use of the schema-free properties of ClassAds by inserting custom attributes into the job ClassAd. These attributes allow Condor to make planning decisions based upon real-time input from production managers, who can tag a project, or a shot, or individual animator with a priority. When jobs are preempted due to changing priorities, Condor will preempt jobs in such a way that minimizes the loss of forward progress as defined by C.O.R.E.'s policy expressions.

To date, Condor has been used by C.O.R.E. for many major productions such as *X-Men*, *Blade II*, *Nutty Professor II*, and *The Time Machine*.

Academic Case Study: NUG30 Optimization Challenge

In the summer of year 2000, four mathematicians from Argonne National Laboratory, University of Iowa, and Northwestern University used Condor-G and several other technologies discussed in this document to be the first to solve a problem known as NUG30. [16] NUG30 is a quadratic assignment problem that was first proposed in 1968 as one of the most difficult combinatorial optimization challenges, but remained unsolved for 32 years because of its complexity.

In order to solve NUG30, the mathematicians started with a sequential solver based upon a *branch-and-bound* tree search technique. This technique divides the initial search space into smaller pieces and bounds what could be the best possible solution in each of these smaller regions. Although the sophistication level of the solver was enough to drastically reduce the amount of compute time it would take to determine a solution, the amount of time was still considerable: over seven years with the best desktop workstation available to the researchers at that time (a Hewlett Packard C3000).

To combat this computation hurdle, a parallel implementation of the solver was developed which fit the master-worker model. The actual computation itself was managed by Condor's *Master-Worker* (MW) problem solving environment. MW submitted work to Condor-G, which provided compute resources from around the world by both *direct flocking* to other Condor pools and by *gliding in* to other compute resources accessible via the Globus GRAM protocol. *Remote System Calls*, part of Condor's *standard universe*, was used as the I/O service between the master and the workers. *Checkpointing* was performed every 15 minutes for fault tolerance. All of these technologies were introduced earlier in this paper.

The end result: a solution to NUG30 was discovered utilizing Condor-G in a computational run of less than one week. During this week, over 95,000 CPU hours were used to solve the over 540 billion linear assignment problems necessary to crack NUG30. Condor-G allowed the mathematicians to harness over 2500 CPUs at ten different sites (eight Condor pools, one compute cluster managed by PBS, and one supercomputer managed by LSF) spanning eight different institutions. Additional statistics about the NUG30 run are presented in Table I.

Conclusion

Through its lifetime, the Condor software has grown in power and flexibility. As other systems such as Kerberos, PVM, and Java have reached maturity and widespread deployment, Condor has adjusted to accommodate the needs of users and administrators without sacrificing its essential design. In fact, the Condor kernel shown in Figure 3 has not changed at all since 1988. Why is this?

We believe the key to lasting system design is to outline structures first in terms of *responsibility* rather than expected *functionality.* This may lead to interactions which, at first blush, seem complex. Consider, for example, the four steps to matchmaking shown in Figure 11 or the six steps to accessing a file shown in Figures 16 and 17. Yet, every step is necessary for discharging a component's responsibility. The matchmaker is responsible for enforcing community policies, so the agent cannot claim a resource without its blessing. The shadow is responsible for enforcing the user's policies, so the sandbox cannot open a file without its help. The apparent complexity preserves the independence of each component. We may update one with more complex policies and mechanisms without harming another.

The Condor project will also continue to grow. The project is home to a variety of systems research ventures in addition to the flagship Condor software. These include the Bypass [75] toolkit, the ClassAd [61] resource management language, the Hawkeye [5] cluster management system, the NeST storage appliance [19]. and the Public Key Infrastructure Lab. [11] In these and other ventures, the project seeks to gain the hard but valuable experience of nurturing research concepts into production software. To this end, the project is a key player in collaborations such as the National Middleware Initiative (NMI) [9] that aim to harden and disseminate research systems as stable tools for end users. The project will continue to train students, solve hard problems, and accept and integrate good solutions from others.

We look forward to the challenges ahead!

ACKNOWLEDGEMENTS

We would like to acknowledge all of the people who have contributed to the development of the Condor system over the years. They are too many to list here, but include faculty and staff; graduates and undergraduates; visitors and residents. However, we must particularly recognize the first core architect of Condor, Mike Litzkow, whose guidance through example and advice has deeply influenced the Condor software and philosophy.

We are also grateful to Brooklin Gore and Doug Warner at Micron Technology, and to Mark Visser at C.O.R.E. Digital Pictures for their Condor enthusiasm and for sharing their experiences with us. Jamie Frey, Mike Litzkow, and Alain Roy provided sound advice as this chapter was written. Thank you!

The Condor project is supported by a wide variety of funding sources. In addition, Douglas Thain is supported in part by a Cisco distinguished graduate fellowship and a Lawrence Landweber NCR fellowship in distributed systems.

REFERENCES

1. European Union DataGrid Project. http://www.eu-datagrid.org.
2. Alliance Partners for Advanced Computational Services (PACS). http://www.ncsa.uiuc.edu/About/Alliance/Teams, 2002.
3. The Compact Muon Solenoid Collaboration. http://uscms.fnal.gov, August 2002.
4. The Grid Physics Network (GriPhyN). http://www.griphyn.org, August 2002.
5. HawkEye: A monitoring and management tool for distributed systems. http://www.cs.wisc.edu/condor/hawkeye, 2002.
6. The International Virtual Data Grid Laboratory (iVDGL). http://www.ivdgl.org, August 2002.
7. NASA Information Power Grid. http://www.ipg.nasa.gov/, August 2002.
8. The National Computational Science Alliance. http://www.ncsa.uiuc.edu/About/Alliance/, August 2002.
9. NSF Middleware Initiative (NMI). http://www.nsf-middleware.org, August 2002.
10. Particle Physics Data Grid (PPDG). http://www.ppdg.net, August 2002.
11. Public Key Infrastructure Lab (PKI-Lab). http://www.cs.wisc.edu/pkilab, August 2002.
12. TeraGrid Project. http://www.teragrid.org, August 2002.
13. William Allcock, Ann Chervenak, Ian Foster, Carl Kesselman, and Steve Tuecke. Protocols and services for distributed data-intensive science. In *Proceedings of Advanced Computing and Analysis Techniques in Physics Research (ACAT)*, pages 161–163, 2000.
14. D. Anderson, S. Bowyer, J. Cobb, D. Gedye, W.T. Sullivan, and D. Werthimer. Astronomical and biochemical origins and the search for life in the universe. In *Proceedings of the 5th International Conference on Bioastronomy*. Editrice Compositori, Bologna, Italy, 1997.
15. D. Angulo, I. Foster, C. Liu, , and L. Yang. Design and evaluation of a resource selection framework for grid applications. In *Proceedings of the 11th IEEE Symposium on High Performance Distributed Computing (HPDC-11)*, Edinburgh, Scotland, July 2002.
16. K. Anstreicher, N. Brixius, J.-P. Goux, and J. Linderoth. Solving large quadratic assignment problems on computational grids. In *Mathematical Programming*, 2000.
17. Jim Basney and Miron Livny. Deploying a high throughput computing cluster. In Rajkumar Buyya, editor, *High Performance Cluster Computing: Architectures and Systems, Volume 1*. Prentice Hall PTR, 1999.

18. Jim Basney, Miron Livny, and Paolo Mazzanti. Utilizing widely distributed computational resources efficiently with execution domains. *Computer Physics Communications*, 2001.
19. John Bent, Venkateshwaran Venkataramani, Nick LeRoy, Alain Roy, Joseph Stanley, Andrea Arpaci Dusseau, Remzi Arpaci-Dusseau, and Miron Livny. Flexibility, manageability, and performance in a grid storage appliance. In *Proceedings of the Eleventh IEEE Symposium on High Performance Distributed Computing*, Edinburgh, Scotland, July 2002.
20. B.N. Bershad, S. Savage, P.Pardyak, E.G. Sirer, M. Fiuchynski, D. Becker, S. Eggers, and C. Chambers. Extensibility, safety, and performance in the SPIN operating system. In *Proceedings of the 15th ACM Symposium on Operating Systems Principles*, pages 251–266, December 1995.
21. Andrew D. Birrell, Roy Levin, Roger M. Needham, and Michael D. Schroeder. Grapevine: An exercise in distributed computing. *Communications of the ACM*, 25(4):260–274, April 1982.
22. Alan Bricker, Mike Litzkow, and Miron Livny. Condor technical summary. Technical Report 1069, University of Wisconsin, Computer Sciences Department, January 1992.
23. R.M. Bryant and R.A. Finkle. A stable distributed scheduling algorithm. In *Proceedings of the Second International Conference on Distributed Computing Systems*, pages 314–323, Paris, France, April 1981.
24. K. Chandy and L. Lamport. Distributed snapshots: Determining global states of distributed systems. *ACM Transactions on Computer Systems*, 3(1):63–75, 1985.
25. Chungmin Chen, Kenneth Salem, and Miron Livny. The DBC: Processing scientific data over the internet. In *16th International Conference on Distributed Computing Systems*, May 1996.
26. Y.C. Chow and W.H. Kohler. Dynamic load balancing in homogeneous two-processor distributed systems. In *Proceedings of the International Symposium on Computer Performance, Modeling, Measurement and Evaluation*, pages 39–52, Yorktown Heights, New York, August 1977.
27. I.B.M. Corporation. *IBM Load Leveler: User's Guide*. I.B.M. Corporation, September 1993.
28. K. Czajkowski, I. Foster, N. Karonis, C. Kesselman, S. Martin, W. Smith, and S. Tuecke. A resource management architecture for metacomputing systems. In *Proceedings of the IPPS/SPDP Workshop on Job Scheduling Strategies for Parallel Processing*, pages 62–82, 1988.
29. David DeWitt, Raphael Finkel, and Marvin Solomon. The CRYSTAL multicomputer: Design and implementation experience. *IEEE Transactions on Software Engineering*, September 1984.
30. P.H. Enslow. What is a distributed data processing system? *Computer*, 11(1):13–21, January 1978.
31. D.H.J. Epema, M. Livny, R. van Dantzig, X. Evers, and J. Pruyne. A worldwide flock of Condors: Load sharing among workstation clusters. *Future Generation Computer Systems*, 12:53–65, 1996.
32. C. Anglano et al. Integrating Grid tools to build a Computing Resource Broker: Activities of DataGrid WP1. In *Proceedings of the Conference on Computing in High Energy Physics 2001 (CHEP01)*, Beijing, September 2001.
33. Fritz Ferstl. Job and resource management systems. In Rajkumar Buyya, editor, *High Performance Cluster Computing: Architectures and Systems, Volume 1*. Prentice Hall PTR, 1999.
34. I. Foster, C. Kesselman, G. Tsudik, and S. Tuecke. A security architecture for computational grids. In *Proceedings of the 5th ACM Conference on Computer and Communications Security Conference*, pages 83–92, 1998.
35. Ian Foster and Carl Kesselman. Globus: A metacomputing intrastructure toolkit. *International Journal of Supercomputer Applications*, 11(2):115–128, 1997.
36. Ian Foster and Carl Kesselman. The Globus project: A status report. In *Proceedings of the Seventh Heterogeneous Computing Workshop*, pages 4–19, March 1998.
37. Ian Foster and Carl Kesselman, editors. *The Grid: Blueprint for a New Computing Infrastructure*. Morgan Kaufmann, 1998.
38. James Frey, Todd Tannenbaum, Ian Foster, Miron Livny, and Steve Tuecke. Condor-G: A computation management agent for multi-institutional grids. In *Proceedings of the Tenth IEEE Symposium on High Performance Distributed Computing (HPDC)*, pages 7–9, San Francisco, California, August 2001.
39. James Frey, Todd Tannenbaum, Ian Foster, Miron Livny, and Steve Tuecke. Condor-G: A computation management agent for multi-institutional grids. *Cluster Computing*, 5:237–246, 2002.
40. Jon Giffin, Somesh Jha, and Bart Miller. Detecting manipulated remote call streams. In *Proceedings of the 11th USENIX Security Symposium*, San Francisco, California, August 2002.
41. R. Henderson and D. Tweten. Portable batch system: External reference specification. Technical report, NASA, Ames Research Center, 1996.
42. Cray Inc. Introducing NQE. Technical Report 2153_2.97, Cray Inc., Seattle, WA, February 1997.
43. D. Jackson, Q. Snell, and M. Clement. Core algorithms of the maui scheduler. In *Proceedings of the 7th Workshop on Job Scheduling Strategies for Parallel Processing*, 2001.

44. Phillip E. Krueger. Distributed scheduling for a changing environment. Technical Report UW-CS-TR-780, University of Wisconsin - Madison Computer Sciences Department, June 1988.
45. L. Lamport, R. Shostak, and M. Pease. The byzantine generals problem. *ACM Transactions on Programming Languages and Systems*, 4(3):382–402, July 1982.
46. Leslie Lamport. Time, clocks, and the ordering of events in a distributed system. *Communications of the ACM*, 7(21):558–565, July 1978.
47. Butler Lampson. Protection. In *Proceedings of the 5th Princeton Symposium on Information Sciences and Systems*, Princeton University, March 1971.
48. Hugh C. Lauer. Observations on the development of an operating system. In *Proceedings of the 8th Symposium on Operating Systems Principles (SOSP)*, pages 30–36, 1981.
49. Jeff Linderoth, Sanjeev Kulkarni, Jean-Pierre Goux, and Michael Yoder. An enabling framework for master-worker applications on the computational grid. In *Proceedings of the Ninth IEEE Symposium on High Performance Distributed Computing (HPDC9)*, pages 43–50, Pittsburgh, Pennsylvania, August 2000.
50. Michael Litzkow and Miron Livny. Experience with the Condor distributed batch system. In *Proceedings of the IEEE Workshop on Experimental Distributed Systems*, October 1990.
51. Michael Litzkow, Todd Tannenbaum, Jim Basney, and Miron Livny. Checkpoint and migration of UNIX processes in the Condor distributed processing system. Technical Report UW-CS-TR-1346, University of Wisconsin - Madison Computer Sciences Department, April 1997.
52. Michael J. Litzkow. Remote UNIX - Turning idle workstations into cycle servers. In *Proceedings of USENIX*, pages 381–384, Summer 1987.
53. Miron Livny. *The Study of Load Balancing Algorithms for Decentralized Distributed Processing Systems*. PhD thesis, Weizmann Institute of Science, 1983.
54. Miron Livny and Rajesh Raman. High-throughput resource management. In Ian Foster and Carl Kesselman, editors, *The Grid: Blueprint for a New Computing Infrastructure*. Morgan Kaufmann, 1998.
55. Barton P. Miller, Mihai Christodorescu, Robert Iverson, Tevfik Kosar, Alexander Mirgordskii, and Florentina Popovici. Playing inside the black box: Using dynamic instrumentation to create security holes. *Parallel Processing Letters*, 11(2/3), June/September 2001.
56. R. M. Needham. Systems aspects of the cambridge ring. In *Proceedings of the seventh symposium on Operating systems principles*, pages 82–85, 1979.
57. E.I. Organick. *The MULTICS system: An examination of its structure*. The MIT Press, Cambridge, Massachusetts and London, England, 1972.
58. James C. Pruyne. *Resource Management Services for Parallel Applications*. PhD thesis, University of Wisconsin, 1996.
59. Jim Pruyne and Miron Livny. Providing resource management services to parallel applications. In *Proceedings of the Second Workshop on Environments and Tools for Parallel Scientific Computing*, May 1994.
60. Jim Pruyne and Miron Livny. Interfacing Condor and PVM to harness the cycles of workstation clusters. *Future Generation Computer Systems*, 12:67–86, May 1996.
61. Rajesh Raman. *Matchmaking Frameworks for Distributed Resource Management*. PhD thesis, University of Wisconsin, October 2000.
62. Rajesh Raman, Miron Livny, and Marvin Solomon. Matchmaking: Distributed resource management for high throughput computing. In *Proceedings of the Seventh IEEE International Symposium on High Performance Distributed Computing (HPDC7)*, July 1998.
63. Rajesh Raman, Miron Livny, and Marvin Solomon. Resource management through multilateral matchmaking. In *Proceedings of the Ninth IEEE Symposium on High Performance Distributed Computing (HPDC9)*, pages 290–291, Pittsburgh, PA, August 2000.
64. J.H. Saltzer and M.D. Schroeder. The protection of information in computer systems. *Proceedings of the IEEE*, 63(9):1278–1308, September 1975.
65. M.I. Seltzer, Y. Endo, C. Small, and K. A. Smith. Dealing with disaster: Surviving misbehaved kernel extensions. In *Proceedings of the 2nd USENIX Symposium on Operating Systems Design and Implementation (OSDI)*, pages 213–227, October 1996.
66. Marvin Solomon and Michael Litzkow. Supporting checkpointing and process migration outside the UNIX kernel. In *Proceedings of USENIX*, pages 283–290, Winter 1992.
67. T. Sterling, D. Savarese, D. J. Becker, J. E. Dorband, U. A. Ranawake, and C. V. Packer. BEOWULF: A parallel workstation for scientific computation. In *Proceedings of the 24th International Conference on Parallel Processing*, pages 11–14, Oconomowoc, Wisconsin, 1995.

68. Richard Stern. Micro law: Napster: A walking copyright infringement? *IEEE Micro*, 20(6):4–5, November/December 2000.
69. H.S. Stone. Multiprocessor scheduling with the aid of network flow algorithms. *IEEE Trans of Software Engineering*, SE-3(1):95–93, January 1977.
70. Todd Tannenbaum, Derek Wright, Karen Miller, and Miron Livny. Condor – A distributed job scheduler. In Thomas Sterling, editor, *Beowulf Cluster Computing with Linux*. MIT Press, October 2001.
71. Todd Tannenbaum, Derek Wright, Karen Miller, and Miron Livny. Condor – A distributed job scheduler. In Thomas Sterling, editor, *Beowulf Cluster Computing with Windows*. MIT Press, October 2001.
72. Condor Team. Condor Manual. Available from `http://www.cs.wisc.edu/condor/manual`, 2001.
73. Douglas Thain, Jim Basney, Se-Chang Son, and Miron Livny. The Kangaroo approach to data movement on the grid. In *Proceedings of the Tenth IEEE Symposium on High Performance Distributed Computing (HPDC10)*, San Francisco, California, August 2001.
74. Douglas Thain, John Bent, Andrea Arpaci-Dusseau, Remzi Arpaci-Dusseau, and Miron Livny. Gathering at the well: Creating communities for grid I/O. In *Proceedings of Supercomputing 2001*, Denver, Colorado, November 2001.
75. Douglas Thain and Miron Livny. Multiple bypass: Interposition agents for distributed computing. *Journal of Cluster Computing*, 4:39–47, 2001.
76. Douglas Thain and Miron Livny. Error scope on a computational grid: Theory and practice. In *Proceedings of the 11th IEEE Symposium on High Performance Distributed Computing (HPDC)*, July 2002.
77. Sudharshan Vazhkudai, Steven Tuecke, and Ian Foster. Replica selection in the globus data grid. *IEEE International Symposium on Cluster Computing and the Grid (CCGrid)*, pages 106–113, May 2001.
78. Bruch Walker, Gerald Popek, Robert English, Charles Kline, and Greg Thiel. The LOCUS distributed operating system. In *Proceedings of the 9th Symposium on Operating Systems Principles (SOSP)*, pages 49–70, November 1983.
79. Derek Wright. Cheap cycles from the desktop to the dedicated cluster: combining opportunisitc and dedicated scheduling with Condor. In *Conference on Linux Clusters: The HPC Revolution*, Champaign-Urbana, Illinois, June 2001.
80. Victor C. Zandy, Barton P. Miller, and Miron Livny. Process hijacking. In *Proceedings of the 8th IEEE International Symposium on High Performance Distributed Computing (HPDC8)*, 1999.
81. S. Zhou. LSF: Load sharing in large-scale heterogenous distributed systems. In *Proc. Workshop on Cluster Computing*, 1992.

