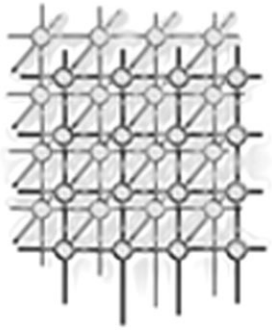
CONCURRENCY AND COMPUTATION: PRACTICE AND EXPERIENCE
Concurrency Computat.: Pract. Exper. 2005; **17**:323–356
Published online in Wiley InterScience (www.interscience.wiley.com). DOI: 10.1002/cpe.938

Distributed computing in practice: the Condor experience



Douglas Thain*,†, Todd Tannenbaum and Miron Livny

Computer Sciences Department, University of Wisconsin-Madison, 1210 West Dayton Street, Madison, WI 53706, U.S.A.

SUMMARY

Since 1984, the Condor project has enabled 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 paper, 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 correspond to social structures. Throughout, we reflect on the lessons of experience and chart the course travelled by research ideas as they grow into production systems. Copyright © 2005 John Wiley & Sons, Ltd.

KEY WORDS: Condor; Grid; history; community; planning; scheduling; split execution

1. INTRODUCTION

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 users and system designers [1]. 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 [2–4] became so popular that there was even once a minor controversy over the meaning of the word 'distributed' [5].

As this early work made it clear that distributed computing was *feasible*, researchers began to take notice that distributed computing would be *difficult*. When messages may be lost, corrupted, or delayed, robust algorithms must be used in order to build a coherent (if not controllable) system [6–9]. Such lessons were not lost on the system designers of the early 1980s. Production systems such as Locus [10] and Grapevine [11] wrestled with the fundamental tension between consistency, availability, and performance in distributed systems.

*Correspondence to: Douglas Thain, Computer Sciences Department, University of Wisconsin-Madison, 1210 West Dayton Street, Madison, WI 53706, U.S.A.
†E-mail: thain@cs.wisc.edu

Received 29 March 2004
Accepted 11 August 2004

In this environment, the Condor project was born. At the University of Wisconsin, Miron Livny combined his doctoral thesis on cooperative processing [12] with the powerful Crystal Multicomputer [13] designed by Dewitt, Finkel, and Solomon and the novel Remote Unix [14] software designed by Michael 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.

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 [15]. 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. This led to a longstanding Condor motto: *leave the owner in control, regardless of the cost.* A fixed schema for representing users and machines was in constant change and so eventually led to the development of a schema-free resource allocation language called ClassAds [16–18]. It has been observed that most complex systems struggle through an adolescence of five to seven years [19]. Condor was no exception.

Scientific interests began to recognize that coupled commodity machines were significantly less expensive than supercomputers of equivalent power [20]. A wide variety of powerful batch execution systems such as LoadLeveler [21] (a descendant of Condor), LSF [22], Maui [23], NQE [24], and PBS [25] 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 [26,27]. A vision called Grid computing began to build the case for resource sharing across organizational boundaries [28].

Throughout this period, the Condor project immersed itself in the problems of production users. As new programming environments such as PVM [29], MPI [30], and Java [31] 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 [32] and the Particle Physics Data Grid (PPDG) [33], the Condor project took part from initial design to end-user support. As new protocols such as GRAM [34], GSI [35], and GridFTP [36] 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 the features of the system in fine detail. In this paper, we will lay out a broad history of the Condor project and its design philosophy. We 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. Next, we describe how our insistence on dividing responsibility has led to a unique model of cooperative computing called *split execution*. In recent years, the project has added a new focus on data-intensive computing. We outline this new research area and describe out recent contributions. Security has been an increasing user concern over the years. We describe how Condor interacts with a variety of security systems. Finally, we conclude by describing how real users have put Condor to work.

2. THE PHILOSOPHY OF FLEXIBILITY

The Condor design philosophy can be summarized with one word: *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. Four admonitions outline the philosophy of flexibility.

Let communities grow naturally. People have 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. We believe that relationships, obligations, and schemata will develop according to user necessity.

Leave the Owner in control, whatever the cost. 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. Therefore, we must be careful to provide tools for the owner of a resource to set policies and even instantly retract a resource for private use.

Plan without being picky. Progress requires optimism. In a community of sufficient size, there will always be idle resources available to do work. However, 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.

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. 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.

3. THE CONDOR SOFTWARE

Research in distributed computing requires immersion in the real world. To this end, the Condor project maintains, distributes, and supports a variety of computing systems that are deployed by commercial and academic interests world wide. These products are the proving grounds for ideas generated in

the academic research environment. The project is best known for two products: the Condor high-throughput computing system, and the Condor-G agent for Grid computing.

3.1. The Condor high-throughput computing system

Condor is a high-throughput distributed batch computing system. Like other batch systems, Condor provides a job management mechanism, scheduling policy, priority scheme, resource monitoring, and resource management [37,38]. 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 similar to other conventional batch systems, Condor's novel architecture allows it to perform well in environments where other batch systems are weak: *high-throughput computing* and *opportunistic computing*. The goal of a high-throughput computing environment [39] 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 use resources whenever they are available, without requiring 100% availability. The two goals are naturally coupled. High-throughput computing is most easily achieved through opportunistic means.

This requires several unique and powerful tools.

- *ClassAds*. The ClassAd language 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 allocation policy, and to adopt a *planning* approach when incorporating Grid resources. We discuss this approach further 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* [40].
- *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 that 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 tools, Condor can do more than effectively manage dedicated compute clusters [37,38]. 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 on 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,

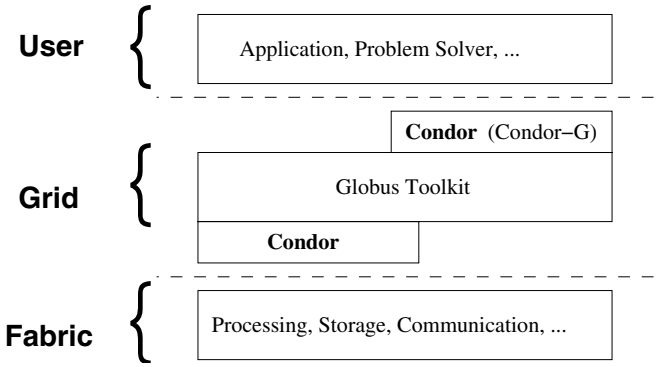


Figure 1. Condor in the Grid.

Condor can preempt any opportunistic computing job that may have been placed there in the meantime [30]. The end result: Condor is used to seamlessly combine all of an organization's computational power into one resource.

3.2. Condor-G: an agent for Grid computing

Condor-G [41] represents the marriage of technologies from the Condor and Globus projects. From Globus [42] 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 a tool that binds resources spread across many systems into a personal high-throughput computing system.

Condor technology can exist at both the front and back ends of a Grid, as depicted in Figure 1. 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 the Globus Toolkit can be used as the bridge between them. In fact, Figure 1 can serve as a simplified diagram for many emerging Grids, such as the USCMS Testbed Grid [43] and the European Union Data Grid [44].

4. AN ARCHITECTURAL HISTORY OF CONDOR

Over the course 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 2. In this section, we examine how a wide variety of *computing communities* may be constructed with small variations to the kernel.

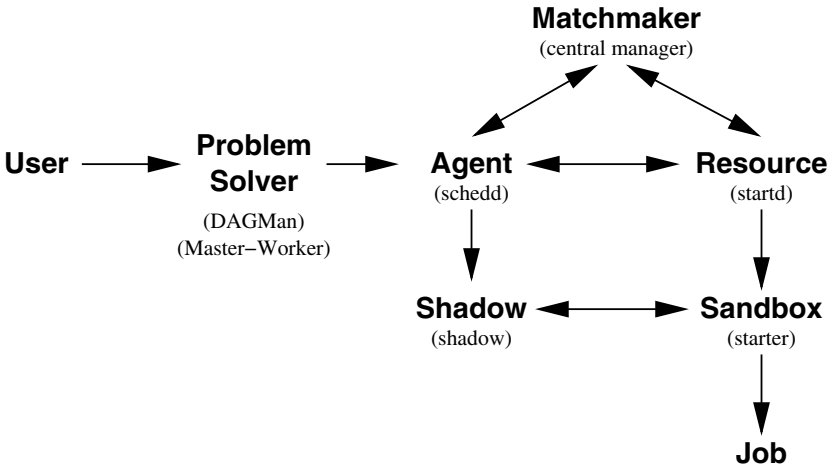


Figure 2. The Condor kernel showing the major processes in a Condor system. The common generic name for each process is given in large print. In parentheses are the technical Condor-specific names used in some publications.

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 paper, we return to examine the other components of the kernel.

The initial conception of Condor is shown in Figure 3. 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 server 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.

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. For example, a user may wish to use machines running the Linux operating system, preferring the use of faster CPUs. The resource enforces the machine owner's policies on what users are to be trusted and serviced. For example, a machine owner might be willing to serve any user, but give preference to members of the Computer Science Department, while rejecting a user known to be untrustworthy. The matchmaker is responsible for enforcing communal policies. For example, a matchmaker might allow any user to access a pool,

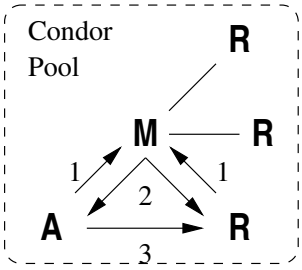


Figure 3. A Condor pool ca. 1988. An agent (A) executes a job on a resource (R) with the help of a matchmaker (M). Step 1: the agent and the resource advertise themselves to the matchmaker. Step 2: the matchmaker informs the two parties that they are potentially compatible. Step 3: the agent contacts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

but limit non-members of the Computer Science Department to consuming ten machines at a time. 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 [45]. 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 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 4. Here, the structure of two existing pools is preserved, while two gateway nodes pass information about participants between the 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.

Gateway flocking was deployed for several years in the 1990s to share computing resources between the United States and several European institutions. Figure 5 shows the state of the world-wide 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.

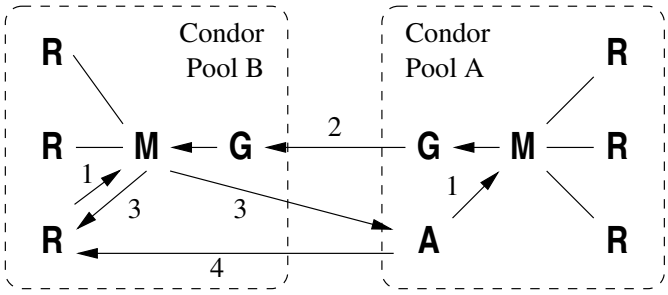


Figure 4. Gateway Flocking ca. 1994. An agent (A) is shown executing a job on a resource (R) via a gateway (G). Step 1: the agent and resource advertise themselves locally. Step 2: the gateway forwards the agent's unsatisfied request to Condor Pool B. Step 3: the matchmaker informs the two parties that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

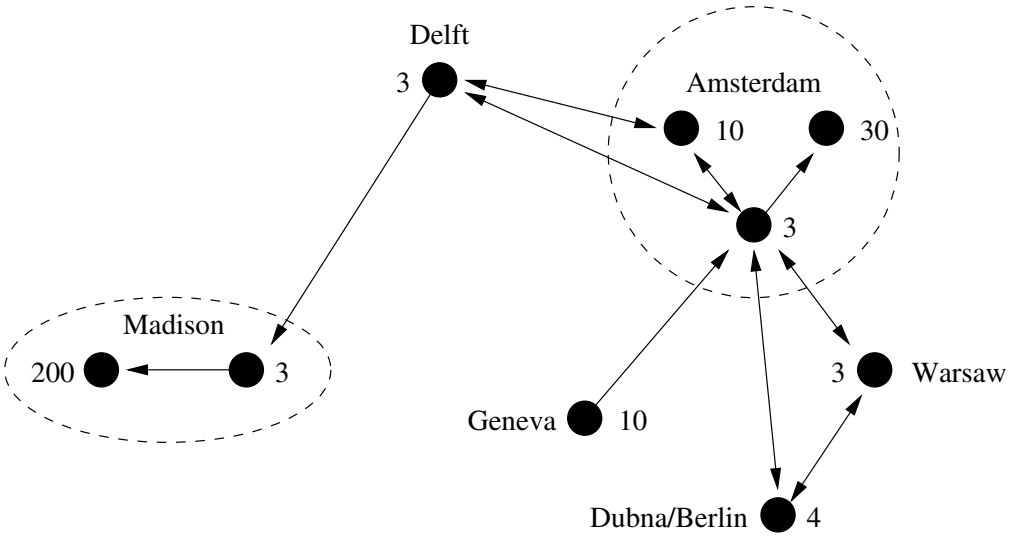


Figure 5. Condor world map ca. 1994. This is a map of the world-wide Condor flock in 1994. Each dot indicates a complete Condor pool. Numbers indicate the size of each Condor pool. Lines indicate flocking via gateways. Arrows indicate the direction that jobs may flow.

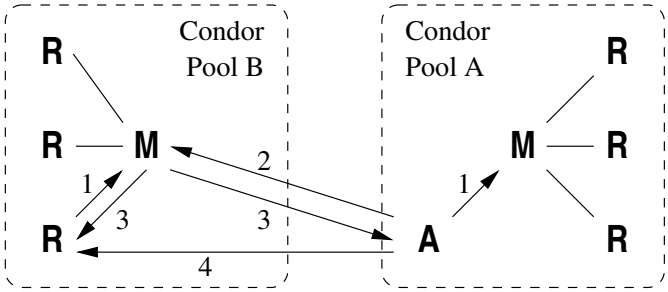


Figure 6. Direct flocking ca. 1998. An agent (A) is shown executing a job on a resource (R) via direct flocking. Step 1: the agent and the resource advertise themselves locally. Step 2: the agent is unsatisfied, so it also advertises itself to Condor Pool B. Step 3: the matchmaker (M) informs the two parties that they are potentially compatible. Step 4: the agent contacts the resource and executes a job.

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 6. 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.

This is a reasonable trade-off 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. However, 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

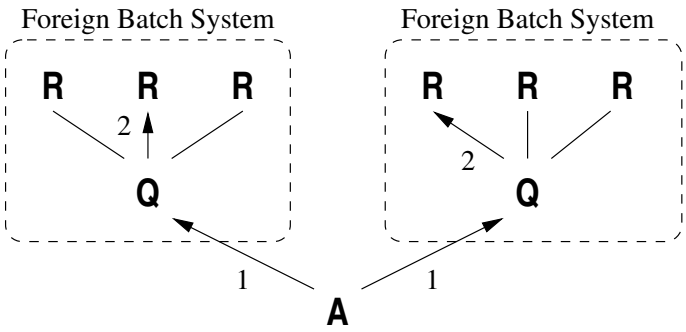


Figure 7. Condor-G ca. 2000. An agent (A) is shown executing two jobs through foreign batch queues (Q). Step 1: the agent transfers jobs directly to remote queues. Step 2: the jobs wait for idle resources (R), and then execute on them.

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.

Around 1998, a vision of a world-wide computational Grid began to grow [28]. A significant early piece in the Grid computing vision was a uniform interface for batch execution. The Globus Project [42] designed the Grid Resource Access and Management (GRAM) protocol [34] 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 queuing, prioritization, logging, and accounting. To provide this service, the Condor project adapted a standard Condor agent to speak GRAM, yielding a system called Condor-G, shown in Figure 7. This required some small changes to GRAM such as adding durability and two-phase commit to prevent the loss or repetition of jobs [46].

GRAM expands the reach of a user to any sort of batch system, whether it runs Condor or another batch system. For example, the solution of the NUG30 [47] quadratic assignment problem relied on the ability of Condor-G to mediate access to over 1000 hosts spread across tens of batch systems on several continents. (We 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 6, 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.

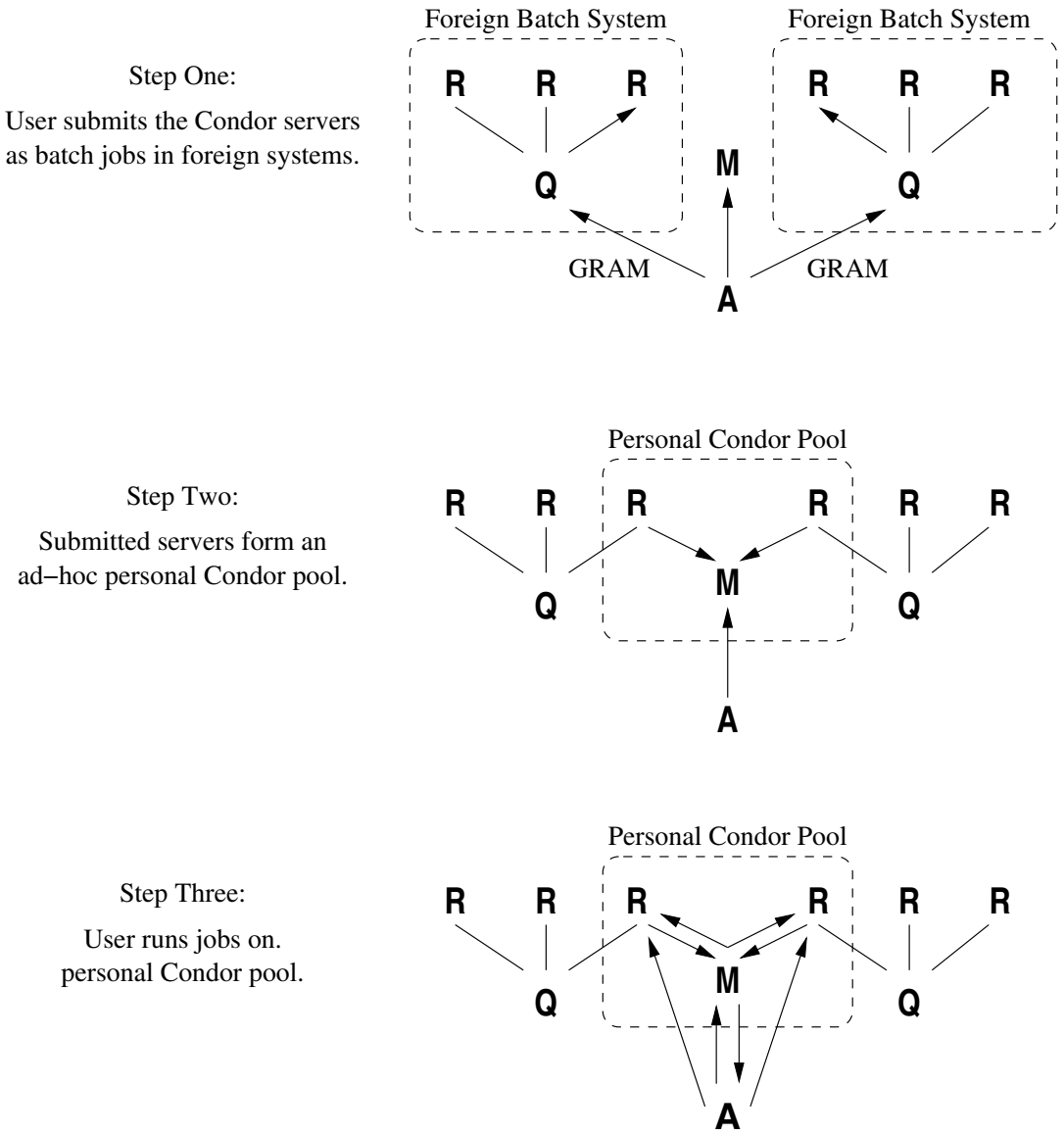


Figure 8. Condor-G and gliding in ca. 2001. A Condor-G agent (A) executes jobs on resources (R) by gliding in through remote batch queues (Q). Step 1: a Condor-G agent submits the Condor servers to two foreign batch queues via GRAM. Step 2: the servers form a personal Condor pool with the user's personal matchmaker (M). Step 3: the agent executes jobs as in Figure 3.

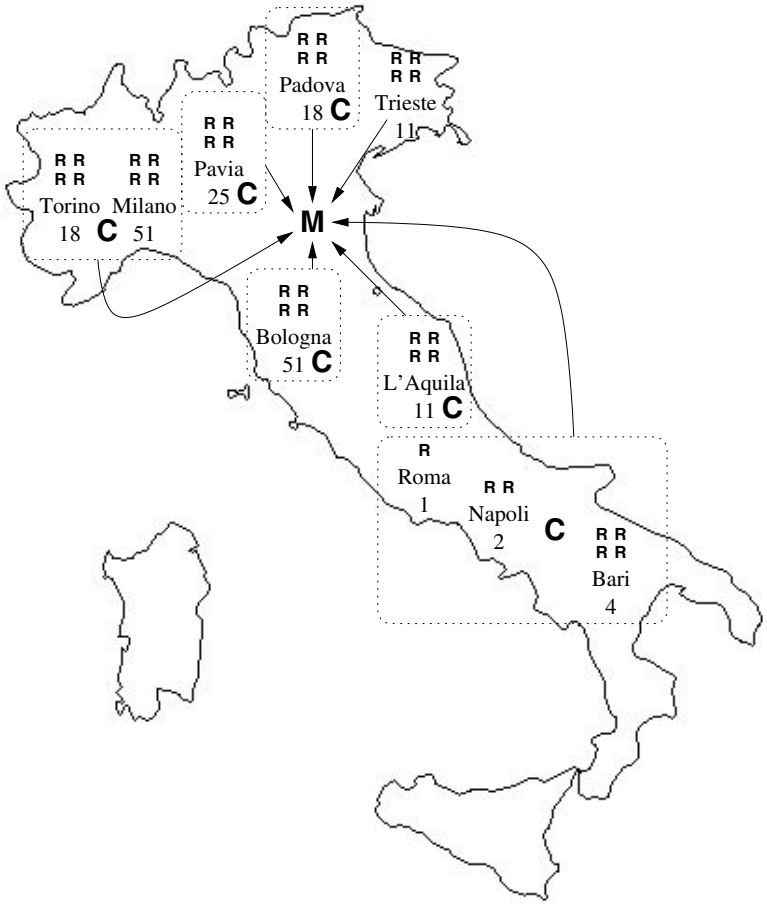


Figure 9. INFN Condor pool ca. 2002. This is a map of a single Condor pool spread across Italy. All resources (R) across the country share the same matchmaker (M) in Bologna. Dotted lines indicate execution domains where resources share a checkpoint server (C). Numbers indicate resources at each site. Resources not assigned to a domain use the checkpoint server in Bologna.

This problem is solved with a technique called *gliding in*, shown in Figure 8. To take advantage of both the powerful reach of GRAM 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 servers as jobs to remote batch systems. From the remote system's perspective, the Condor servers are ordinary jobs with no special privileges. In the second step, the servers 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,

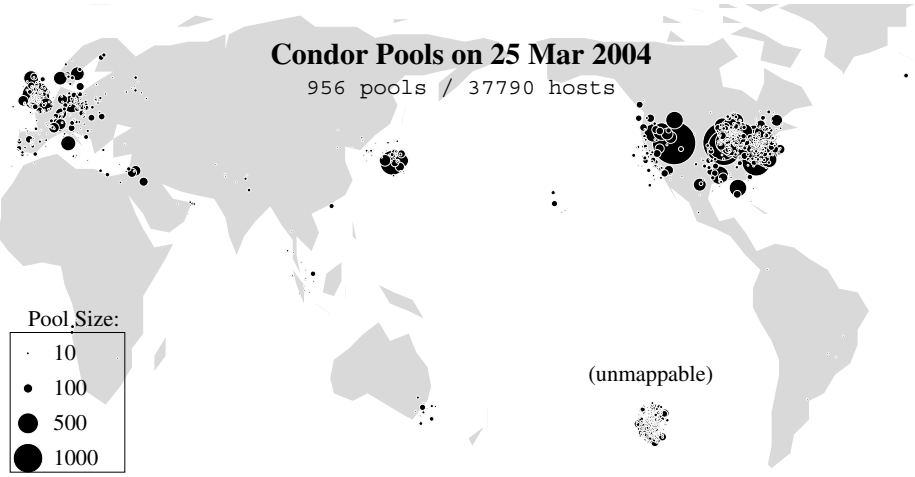


Figure 10. Condor world map ca. 2004. Each dot indicates an independent Condor pool. The area covered by each dot is proportional to the number of machines in the pool. The location of each pool is determined by the top-level country domain name of the matchmaker, if present, or otherwise from public WHOIS records. Each dot is scattered by a small random factor, thus some appear to fall in the sea. A small number of pools that could not be mapped are plotted in the South Pacific.

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) [48]. 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 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 [49].

Figure 9 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 that 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.

Today, Condor is deployed around the world in pools ranging from a handful of CPUs to thousands of CPUs. Each matchmaker periodically sends a message home to the University of Wisconsin by e-mail of UDP, where we maintain a global catalog of Condor pools. Figure 10 plots this data on a map, showing the size and distribution of Condor around the world. Of course, this map is incomplete: some Condor pools are deployed behind firewalls, and some users voluntarily disable the periodic messages. However, it can be seen that Condor is well used today.

5. 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 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. However, 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 allows the resource and agent to independently verify the match [50].

A ClassAd is a set of uniquely named expressions, using a semi-structured data model so that no specific schema is required by the matchmaker. Each named expression is called an *attribute*. Each attribute has a *name* and a *value*. The first version of the ClassAd language allowed simple values such as integers and strings, as well as expressions comprised of arithmetic and logical operators. After gaining experience with ClassAds, we created a second version of the language that permitted richer values and operators permitting complex data structures such as records, lists, and sets.

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 be 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

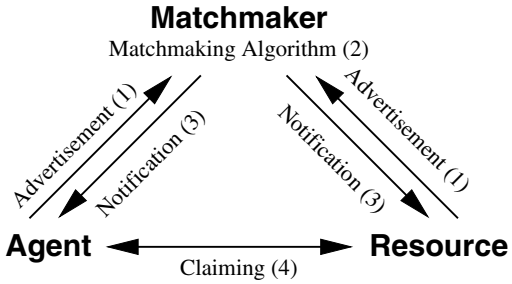


Figure 11. Matchmaking.

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
Crod = "/home/tannenba/bin/sim-exe"	**Arch** = "INTEL"
Department = "CompSci"	**OpSys** = "LINUX"
Owner = "tannenba"	**Disk** = 3076076
DiskUsage = 6000	]
]	

Figure 12. Two sample ClassAds from Condor.

`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 (non-integer 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 MB). 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.

5.1. 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 7). However, if the remote scheduler publishes information about its timetable or workload priorities via a ClassAd to the Condor-G matchmaker, Condor-G could make 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. This approach is used by the PPDG [33]. As more information is published, Condor-G can perform better planning. However, even in the 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 that purchases, in advance, eight seats on a train each week for a year. The company does not control the train schedule, so they must plan how to utilize the buses. However, after purchasing the tickets, the company is free to decide which employees to send to the train station each week. In this manner, Condor performs scheduling within a plan when scheduling parallel jobs on compute clusters [30]. 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 until told otherwise. The agent then creates a schedule for running tasks upon the resources that it has claimed via planning.

5.2. 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* [51] in 1992. By 1995, the expressions had been generalized into *classified advertisements* or ClassAds [52]. 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 [16–18], 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++ [53]. This package enables the matchmaking framework to be used in other distributed computing projects [54,55]. Several research extensions to matchmaking have been built. *Gang matching* [17,18] 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* [56] permits the selection and claiming of large numbers of resource using a very compact expression representation. *Named references* [57] 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 provide requirements and preferences on the resources they wish. However, 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 that 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.

6. PROBLEM SOLVERS

So far, we have delved down into the details of Condor that the user relies on, 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* (MW) and the *Directed Acyclic Graph* (DAG) *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 the 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 that 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.

6.1. 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

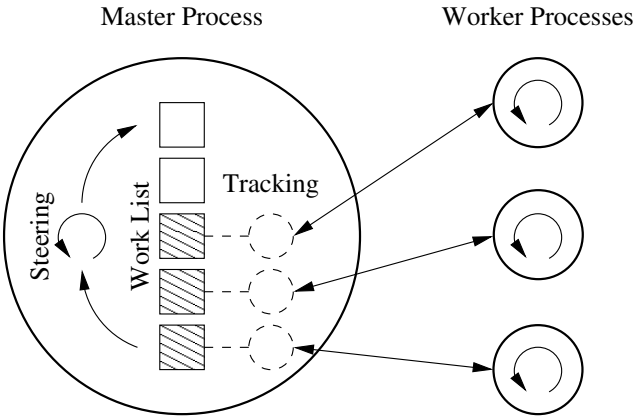


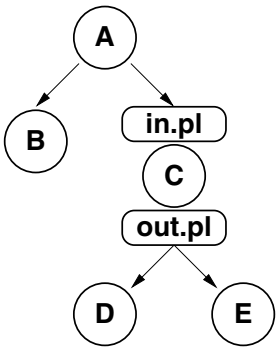
Figure 13. Structure of a MW program.

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. James Pruyne first proposed that applications ought to have an explicit interface for finding resource and placing jobs in a batch system [52]. To facilitate this, such an interface was contributed to the PVM programming environment [58]. The first user of this interface was the Work Distributor (WoDi, pronounced '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 [47]. 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).




```
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.

6.2. DAGMan

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 tool `make`. Like its ancestor, it accepts a declaration that lists the work to be done with constraints on the 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` 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.

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.

7. SPLIT EXECUTION

This far, we have explored the techniques needed merely to get 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 because no-one 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 2 above. Here we 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 work. 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 [59–62], 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 describe the oldest and the newest universes in Condor: the standard universe and the Java universe.

7.1. 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 [14] facility.

The goal of the standard universe is to faithfully reproduce the user's home Unix 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. 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 remote procedure call (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 such as HTTP and NeST [63]. 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

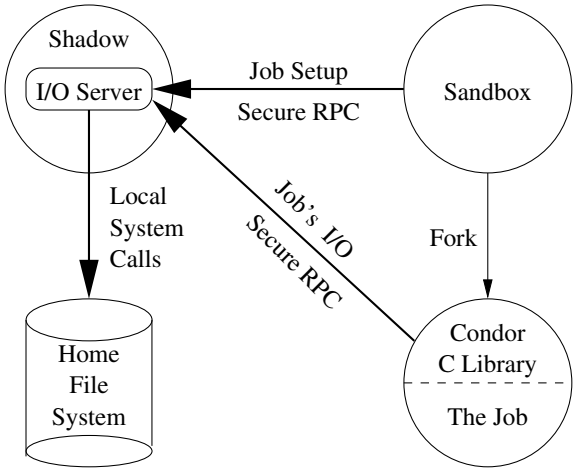


Figure 15. The standard universe.

decisions without the shadow's consent. This maximizes the flexibility of the user to make runtime 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 RPCs, 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.

The mechanics of checkpointing and remote system calls in Condor are described in great detail by Litzkow *et al*. [64,65].

7.2. 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

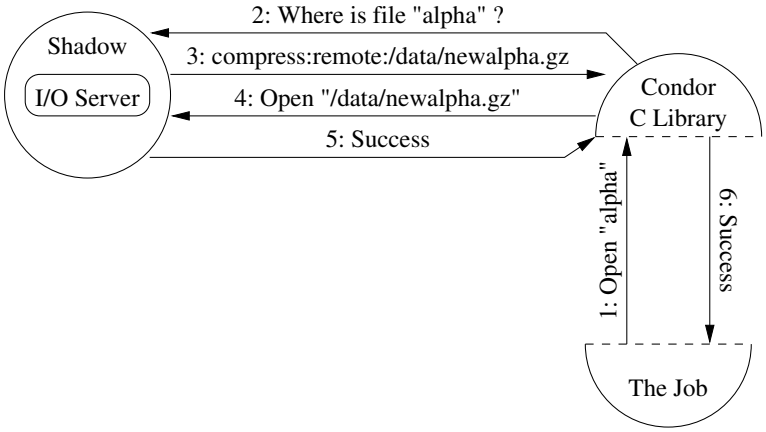


Figure 16. Two-phase open using the shadow.

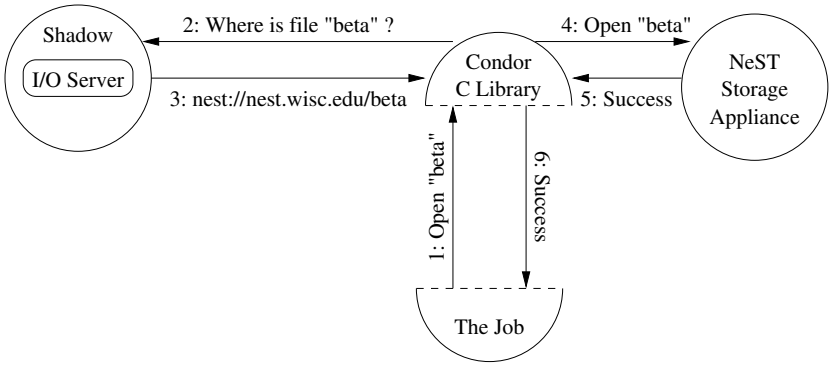


Figure 17. Two-phase open using a nest.

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 that would raise the level of abstraction to create a complete Java environment rather than a Unix environment. The components of the new Java universe are

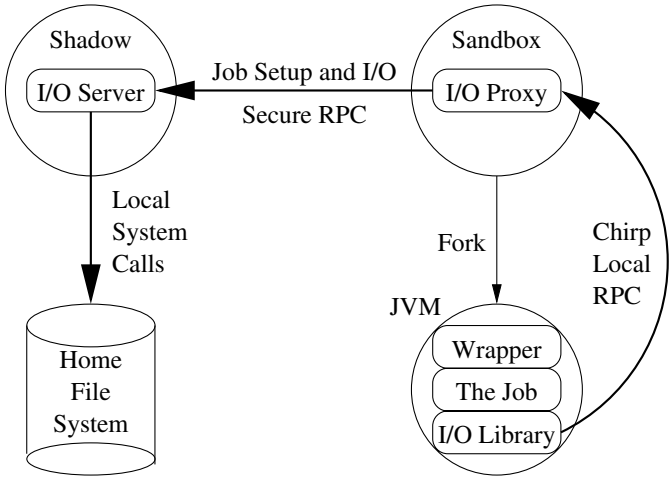


Figure 18. The Java universe.

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.

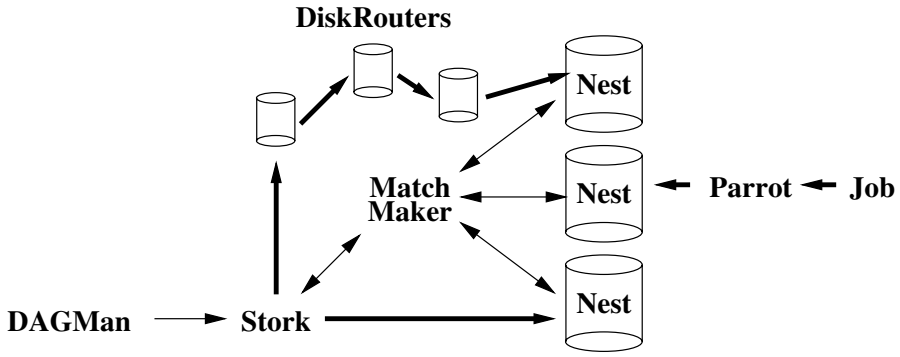


Figure 19. Data-intensive computing tools.

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 [31].

8. DATA-INTENSIVE COMPUTING

Condor was initially conceived as a manager of computing resources, with some incidental facilities for accessing small amounts of data. However, the data needs of batch computing users have grown dramatically in the last two decades. Many forms of science are increasingly centered around large data sets such as image libraries, biological genomes, or simulation results. In the last few years, the Condor project has built a variety of tools for managing and accessing data within a batch system. These tools, shown in Figure 19, are naturally analogous to the Condor kernel described earlier. Because these tools are relative newcomers to the project, we give a high-level overview of their purpose and role, and direct the reader to more detailed publications.

The basic resource to be managed in a data-intensive system is storage. A resource manager called *Nest* supervises a storage device, periodically advertising itself to a matchmaker so that an external user may discover and harness it. Once discovered, Nest-managed storage may be allocated for a given amount of time. A user may then access this allocated space using a variety of standard protocols such as FTP and HTTP [63].

The transfer of data to a remote Nest may be thought of as a batch job much like a program to be run: a transfer must be named, queued, scheduled, logged, and perhaps retried if it fails. A user with many

transfers to carry out as part of a batch workload may queue such transfers with a *Stork*. Like the agent process in standard Condor, a Stork negotiates with a matchmaker, communicates with remote storage devices, and generally makes large data transfers reliable. A Stork speaks many standard protocols such as FTP, HTTP, and Nest, and thus may arrange transfers between many types of devices. The choice of a protocol is dynamically chosen to maximize throughput [66].

Very large data transfers over the wide area are often adversely affected by fluctuating local network congestion. To smooth out large transfers, a series of *DiskRouters* can serve as in-network buffering space for transfers between storage devices. When arranging a transfer, a Stork may arrange for one or more DiskRouters to 'smooth out' local bandwidth fluctuations and thus achieve higher end-to-end bandwidth [67].

In this heterogeneous environment, a job's data may be stored in a wide variety of devices, each with their own peculiar naming scheme and access protocol. *Parrot* connects ordinary programs to these unusual devices. Parrot uses the debugger interface to trap a program's system calls and convert them into operations on remote storage devices, which simply appear as entries in the filesystem. In this way, ordinary programs and scripts can be deployed into this system without any changes. As with the other components, a variety of protocols are supported [68].

Of course, the Condor data components must work in concert with the computation components. Coordination can be achieved at a high level by using the DAGMan problem solver. In the same manner that DAGMan can dispatch ready jobs to a Condor agent, it can also dispatch data placement requests to a Stork. In this manner, an entire DAG can be constructed that stages data to a remote site, runs a series of jobs, retrieves the output, and cleans up the space.

The project is only just beginning to develop experience with deploying and using these tools. Management of data is a large and multi-faceted problem that we will continue to explore in the future.

9. SECURITY

Condor shares many of the same security challenges as other distributed computing environments, and the notion of split-execution adds several more. The security mechanisms in Condor strive to provide flexible secure communication, protect resource owners from errant or malicious jobs, and protect Condor users from errant or malicious resources.

9.1. Secure communication

Secure communication in Condor is handled by *CEDAR*, a message-based communication library developed by the Condor Project. This library allows clients and servers to negotiate and use a variety of security protocols. This ability is critical because Condor is deployed at many different sites, each with their own local security mechanism and policy. CEDAR permits authentication via Kerberos [69], GSI [35], 3DES [70], Blowfish [71], and Microsoft's SSPI [72]. For sites that are secured by an external device such as a firewall, CEDAR also permits simple authentication via trusted hosts or networks. CEDAR bears a similarity to SASL [73], but also supports connection-based and connectionless (datagram) communications, as well as the ability to negotiate data integrity and privacy algorithms separately from the authentication protocol.

Although currently Condor's secure communication model is based upon a secure connection established via CEDAR, we are currently investigating message-based security models via digital signatures of ClassAd attributes. With message-based security, authenticity is attached to the payload itself, instead of being attached to the communication channel. A clear advantage of message-based security is that any information exchanged continues to be secure after the network channel has been torn down, permitting data transferred to be stored persistently to disk or transferred via multiple connections and still remain authentic.

9.2. Secure execution

In general, Condor assumes that a fair amount of trust exists between machine owners and users that wish to run jobs. These two parties typically have some relationship, whether they are employees of the same organization or have agreed to collaborate on a specific project. Condor allows owners and users to specify that parties are trusted, but in general, a user must have some external reason to believe that a machine will execute the computations that have been requested. That said, Condor provides mechanisms that protect both machine owners and users from runaway, crashing, and in some cases, malicious parties. Machine owners are protected by the *sandbox* at the execution site, while users are protected by the *shadow* at the submission site.

The sandbox at the execution site prevents a job from accessing resources that it should not. The sandbox has evolved over several years of Condor, in reaction to tensions between usability and security. Early versions of Condor restricted running jobs to a limited portion of the filesystem using the Unix *chroot* feature. This technique was abandoned when dynamic linking became prevalent, because it made it quite difficult for users to compose jobs that could execute correctly without library support. Today, jobs are run in an unrestricted filesystem, but are given a restricted login account. Filesystem access control lists can then be used to restrict access by remote jobs.

The selection of a restricted account is a tricky problem in itself. In Unix, one may use the standard *nobody* account to run a job with few privileges. However, if a multi-CPU machine were to run multiple jobs at once, they would be able to interfere with each other by virtue of having the same *nobody* user ID. A malicious Condor user could submit jobs that could hijack another user's jobs. (This exploit, and ones similar to it, are described by Miller *et al*. [74].) To prevent this, Condor now dynamically allocates user IDs for each running job. On Unix machines, this requires the administrator to manually set aside certain IDs. On Windows machines, Condor can automatically allocate users on the fly.

Many Condor installations operate in a cluster or workgroup environment where a common user database is shared among a large number of machines. In these environments, users may wish to have their jobs run with normal credentials so that they may access existing distributed filesystems and other resources. To allow this behavior, Condor has the notion of a *user ID domain*: a set of machines known to share the same user database. When running a job, Condor checks to see if the submitting and executing machine are members of the same user ID domain. If so, the job runs with the submitter's user ID. If not, the job is sandboxed as above.

As we develop new mechanisms to permit split-execution without requiring a relink [68,75], we hope to return to the days of only running jobs via a dynamic nobody account. Having a unique account specific to the launch of one job is beneficial for both for ease of administration and for cleanup. Cleanup of the processes left behind for a given job is easy with the dynamic nobody model—simply kill all processes owned by the nobody user. Conversely, cleanup is problematic when running the

job as a real user. Condor cannot just go and kill all processes owned by a given user, because they may not have all been launched by Condor. Sadly, Unix provides no facility for reliably cleaning up a process tree. (The *setsid* feature is only a voluntary grouping.) Note that starting with Windows 2000, Microsoft leaped ahead of Unix in this regard by providing a reliable way to track all the descendants of a process.

10. CASE STUDIES

Today, Condor is used by organizations around the world. Three brief case studies presented below provide a glimpse of the many ways that Condor is deployed in industry and the academia.

10.1. C.O.R.E. Digital Pictures

C.O.R.E. Digital Pictures is a highly successful Toronto-based computer animation studio, co-founded in 1994 by William Shatner 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.

Today, Condor manages a pool at C.O.R.E. consisting of hundreds of Linux and Silicon Graphics machines. The Linux machines are all dual-CPU and mostly reside on the desktops of the 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 a Condor agent on the desktop. On a busy day, C.O.R.E. animators submit over 15 000 jobs to Condor. 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 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 that it subsequently submits to Condor.

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.

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*.

10.2. Micron Technology, Inc.

Micron Technology, Inc., is a world-wide provider of semiconductors. Significant computational analysis is required to tightly control all of the steps of the engineering process, enabling Micron to achieve short cycle times, high yields, low production costs, and die sizes that are some of the smallest in the industry. 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 enterprize offered Micron the opportunity to meet its engineering needs without incurring the cost associated with traditional, dedicated compute resources. So far, Micron has set up 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.

10.3. NUG30 optimization challenge

In the summer of 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 [76]. 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. 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. To combat this computation hurdle, a parallel implementation of the solver was developed which fit the MW model. The actual computation itself was managed by Condor's 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 540 000 000 000 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 Tables I and II.

11. 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 2 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. Every step is necessary for discharging a component's responsibility. The apparent complexity preserves the independence of each component. We may update one with more complex policies and mechanisms without harming another.

Table I. NUG30 computation statistics: number of CPUs utilized at different locations on the Grid during the seven-day NUG30 run.

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

Table II. NUG30 computation statistics: other interesting statistics about the run.

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

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, such as the ClassAd [18] resource management language, the Hawkeye [77] cluster management system, the NeST storage appliance [63]. and the Public Key Infrastructure Lab [78]. 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 [79] 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 paper was written.

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. Organick EI. *The MULTICS System: An Examination of its Structure*. MIT Press: Cambridge, MA, 1972.
2. Stone HS. Multiprocessor scheduling with the aid of network flow algorithms. *IEEE Transactions on Software Engineering* 1977; **3**(1):95–93.
3. Chow YC, Kohler WH. Dynamic load balancing in homogeneous two-processor distributed systems. *Proceedings of the International Symposium on Computer Performance, Modeling, Measurement and Evaluation*, Yorktown Heights, NY, August 1977. Elsevier/North-Holland: Amsterdam, 1977; 39–52.
4. Bryant RM, Finkle RA. A stable distributed scheduling algorithm. *Proceedings of the Second International Conference on Distributed Computing Systems*, Paris, France, April 1981. IEEE Press: Piscataway, NJ, 1981; 314–323.
5. Enslow PH. What is a distributed data processing system? *Computer* 1978; **11**(1):13–21.
6. Lamport L. Time, clocks, and the ordering of events in a distributed system. *Communications of the ACM* 1978; **7**(21):558–565.
7. Lamport L, Shostak R, Pease M. The Byzantine Generals problem. *ACM Transactions on Programming Languages and Systems* 1982; **4**(3):382–402.
8. Chandy K, Lamport L. Distributed snapshots: Determining global states of distributed systems. *ACM Transactions on Computer Systems* 1985; **3**(1):63–75.
9. Needham RM. Systems aspects of the Cambridge ring. *Proceedings of the 7th Symposium on Operating Systems Principles*. ACM Press: New York, 1979; 82–85.
10. Walker B, Popek G, English R, Kline C, Thiel G. The LOCUS distributed operating system. *Proceedings of the 9th Symposium on Operating Systems Principles (SOSP)*, November 1983. ACM Press: New York, 1983; 49–70.
11. Birrell AD, Levin R, Needham RM, Schroeder MD. Grapevine: An exercise in distributed computing. *Communications of the ACM* 1982; **25**(4):260–274.
12. Livny M. The study of load balancing algorithms for decentralized distributed processing systems. *PhD Thesis*, Weizmann Institute of Science, 1983.
13. DeWitt D, Finkel R, Solomon M. The CRYSTAL multicomputer: Design and implementation experience. *IEEE Transactions on Software Engineering* 1984; **13**(8).
14. Litzkow MJ. Remote UNIX—turning idle workstations into cycle servers. *Proceedings of USENIX*. USENIX Association, 1987; 381–384.
15. Litzkow M, Livny M. Experience with the Condor distributed batch system. *Proceedings of the IEEE Workshop on Experimental Distributed Systems*. IEEE Press: Piscataway, NJ, 1990.
16. Raman R, Livny M, Solomon M. Matchmaking: Distributed resource management for high throughput computing. *Proceedings of the 7th IEEE International Symposium on High Performance Distributed Computing (HPDC7)*. IEEE Press: Piscataway, NJ, 1998.
17. Raman R, Livny M, Solomon M. Resource management through multilateral matchmaking. *Proceedings of the 9th IEEE Symposium on High Performance Distributed Computing (HPDC9)*, Pittsburgh, PA, August 2000. IEEE Press: Piscataway, NJ, 290–291.
18. Raman R. Matchmaking frameworks for distributed resource management. *PhD Thesis*, University of Wisconsin, October 2000.
19. Lauer HC. Observations on the development of an operating system. *Proceedings of the 8th Symposium on Operating Systems Principles (SOSP)*. ACM Press: New York, 1981; 30–36.

20. Sterling T, Savarese D, Becker DJ, Dorband JE, Ranawake UA, Packer CV. BEOWULF: A parallel workstation for scientific computation. *Proceedings of the 24th International Conference on Parallel Processing*, Oconomowoc, WI, 1995; 11–14.
21. IBM Corporation. *IBM Load Leveler: User's Guide*. IBM Corporation, 1993.
22. Zhou S. LSF: Load sharing in large-scale heterogenous distributed systems. *Proceedings of the Workshop on Cluster Computing*, 1992.
23. Jackson D, Snell Q, Clement M. Core algorithms of the maui scheduler. *Proceedings of the 7th Workshop on Job Scheduling Strategies for Parallel Processing*, 2001.
24. Cray Inc. Introducing NQE. *Technical Report 2153_2.97*, Cray Inc., Seattle, WA, February 1997.
25. Henderson R, Tweten D. Portable batch system: External reference specification. *Technical Report*, NASA, Ames Research Center, 1996.
26. Anderson D, Bowyer S, Cobb J, Gedye D, Sullivan WT, Werthimer D. Astronomical and biochemical origins and the search for life in the universe. *Proceedings of the 5th International Conference on Bioastronomy*. Editrice Compositori: Bologna, 1997.
27. Stern R. Micro law: Napster: A walking copyright infringement? *IEEE Micro* 2000; **20**(6):4–5.
28. Foster I, Kesselman C (eds.). *The Grid: Blueprint for a New Computing Infrastructure*. Morgan Kaufmann: San Mateo, CA, 1998.
29. Pruyne J, Livny M. Providing resource management services to parallel applications. *Proceedings of the 2nd Workshop on Environments and Tools for Parallel Scientific Computing*, May 1994.
30. Wright D. Cheap cycles from the desktop to the dedicated cluster: Combining opportunistic and dedicated scheduling with Condor. *Conference on Linux Clusters: The HPC Revolution*, Champaign-Urbana, IL, June 2001.
31. Thain D, Livny M. Error scope on a computational Grid: Theory and practice. *Proceedings of the 11th IEEE Symposium on High Performance Distributed Computing (HPDC)*. IEEE Press: Piscataway, NJ, 2002.
32. The Grid Physics Network (GriPhyN). http://www.griphyn.org [August 2002].
33. Particle Physics Data Grid (PPDG). http://www.ppdg.net [August 2002].
34. Czajkowski K, Foster I, Karonis N, Kesselman C, Martin S, Smith W, Tuecke S. A resource management architecture for metacomputing systems. *Proceedings of the IPPS/SPDP Workshop on Job Scheduling Strategies for Parallel Processing*, 1988; 62–82.
35. Foster I, Kesselman C, Tsudik G, Tuecke S. A security architecture for computational Grids. *Proceedings of the 5th ACM Conference on Computer and Communications Security Conference*. ACM Press: New York, 1998; 83–92.
36. Allcock W, Chervenak A, Foster I, Kesselman C, Tuecke S. Protocols and services for distributed data-intensive science. *Proceedings of Advanced Computing and Analysis Techniques in Physics Research (ACAT)*, 2000; 161–163.
37. Tannenbaum T, Wright D, Miller K, Livny M. Condor—a distributed job scheduler. *Beowulf Cluster Computing with Linux*, Sterling T (ed.). MIT Press: Cambridge, MA, 2001.
38. Tannenbaum T, Wright D, Miller K, Livny M. Condor—a distributed job scheduler. *Beowulf Cluster Computing with Windows*, Sterling T (ed.). MIT Press: Cambridge, MA, 2001.
39. Basney J, Livny M. Deploying a high throughput computing cluster. *High Performance Cluster Computing: Architectures and Systems*, vol. 1, Buyya R (ed.). Prentice-Hall: Englewood Cliffs, NJ, 1999.
40. Krueger PE. Distributed scheduling for a changing environment. *Technical Report UW-CS-TR-780*, University of Wisconsin-Madison, Computer Sciences Department, June 1988.
41. Frey J, Tannenbaum T, Foster I, Livny M, Tuecke S. Condor-G: A computation management agent for multi-institutional Grids. *Proceedings of the 10th IEEE Symposium on High Performance Distributed Computing (HPDC)*, San Francisco, CA, August 2001. IEEE Press: Piscataway, NJ, 2001; 7–9.
42. Foster I, Kesselman C. Globus: A metacomputing intrastructure toolkit. *International Journal of Supercomputer Applications* 1997; **11**(2):115–128.
43. The Compact Muon Solenoid Collaboration. http://uscms.fnal.gov [August 2002].
44. European Union DataGrid Project. http://www.eu-dataGrid.org.
45. Epema DHJ, Livny M, van Dantzig R, Evers X, Pruyne J. A world-wide flock of Condors: Load sharing among workstation clusters. *Future Generation Computer Systems* 1996; **12**:53–65.
46. Frey A, Tannenbaum T, Foster I, Livny M, Tuecke S. Condor-G: A computation management agent for multi-institutional Grids. *Cluster Computing* 2002; **5**:237–246.
47. Linderoth J, Kulkarni S, Goux J-P, Yoder M. An enabling framework for master–worker applications on the computational Grid. *Proceedings of the 9th IEEE Symposium on High Performance Distributed Computing (HPDC9)*, Pittsburgh, PA, August 2000. IEEE Press: Piscataway, NJ, 2000; 43–50.
48. Chen C, Salem K, Livny M. The DBC: Processing scientific data over the Internet. *Proceedings of the 16th International Conference on Distributed Computing Systems*, May 1996.
49. Basney J, Livny M, Mazzanti P. Utilizing widely distributed computational resources efficiently with execution domains. *Computer Physics Communications* 2001; **140**(1–2):246.

50. Livny M, Raman R. High-throughput resource management. *The Grid: Blueprint for a New Computing Infrastructure*, Foster I, Kesselman C (eds.). Morgan Kaufmann: San Mateo, CA, 1998.
51. Bricker A, Litzkow M, Livny M. Condor technical summary. *Technical Report 1069*, Computer Sciences Department, University of Wisconsin, January 1992.
52. Pruyne JC. Resource management services for parallel applications. *PhD Thesis*, University of Wisconsin, 1996.
53. Condor Team. Condor Manual. http://www.cs.wisc.edu/condor/manual [2001].
54. Vazhkudai S, Tuecke S, Foster I. Replica selection in the globus data Grid. *Proceedings of the IEEE International Symposium on Cluster Computing and the Grid (CCGrid)*, May 2001; 106–113.
55. Anglano C *et al*. Integrating Grid tools to build a computing resource broker: Activities of DataGrid WP1. *Proceedings of the Conference on Computing in High Energy Physics 2001 (CHEP01)*, Beijing, September 2001.
56. Angulo D, Foster I, Liu C, Yang L. Design and evaluation of a resource selection framework for Grid applications. *Proceedings of the 11th IEEE Symposium on High Performance Distributed Computing (HPDC-11)*, Edinburgh, Scotland, July 2002. IEEE Press: Piscataway, NJ, 2002.
57. Thain D, Bent J, Arpaci-Dusseau A, Arpaci-Dusseau R, Livny M. Gathering at the well: Creating communities for Grid I/O. *Proceedings of Supercomputing 2001*, Denver, CO, November 2001.
58. Pruyne J, Livny M. Interfacing Condor and PVM to harness the cycles of workstation clusters. *Future Generation Computer Systems* 1996; **12**:67–86.
59. Lampson B. Protection. *Proceedings of the 5th Princeton Symposium on Information Sciences and Systems*. Princeton University Press: Princeton, NJ, 1971.
60. Saltzer JH, Schroeder MD. The protection of information in computer systems. *Proceedings of the IEEE* 1975; **63**(9):1278–1308.
61. Bershad BN, Savage S, Pardyak P, Sirer EG, Fiuchynski M, Becker D, Eggers S, Chambers C. Extensibility, safety, and performance in the SPIN operating system. *Proceedings of the 15th ACM Symposium on Operating Systems Principles*, December 1995. ACM Press: New York, 1995; 251–266.
62. Seltzer MI, Endo Y, Small C, Smith KA. Dealing with disaster: Surviving misbehaved kernel extensions. *Proceedings of the 2nd USENIX Symposium on Operating Systems Design and Implementation (OSDI)*, October 1996. USENIX Association, 1996; 213–227.
63. Bent J, Venkataramani V, LeRoy N, Roy A, Stanley J, Dusseau AA, Arpaci-Dusseau R, Livny M. Flexibility, manageability, and performance in a Grid storage appliance. *Proceedings of the 11th IEEE Symposium on High Performance Distributed Computing*, Edinburgh, Scotland, July 2002. IEEE Press: Piscataway, NJ, 2002.
64. Solomon M, Litzkow M. Supporting checkpointing and process migration outside the UNIX kernel. *Proceedings of USENIX*. USENIX Association, 1992; 283–290.
65. Litzkow M, Tannenbaum T, Basney J, Livny M. 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.
66. Kosar T, Livny M. Stork: Making data placement a first class citizen in the Grid. *Proceedings of the 24th IEEE International Conference on Distributed Computing Systems*, Tokyo, Japan, March 2004. IEEE Press: Piscataway, NJ, 2004.
67. Kosar T, Kola G, Livny M. A framework for self-optimising, fault-tolerant, high performance bulk data transfers in a heterogeneous Grid environment. *Proceedings of the 2nd International Symposium on Parallel and Distributed Computing*, Ljubljana, Slovenia, October 2003.
68. Thain D, Livny M. Parrot: Transparent user-level middleware for data-intensive computing. *Proceedings of the Workshop on Adaptive Grid Middleware*, New Orleans, LA, September 2003.
69. Steiner JG, Neuman C, Schiller JI. Kerberos: An authentication service for open network systems. *Proceedings of USENIX*. USENIX Association, 1988; 191–200.
70. National Bureau of Standards. *Data Encryption Standard*. Department of Commerce, Washington, DC, January 1977.
71. Schneier B. The Blowfish encryption algorithm. *Dr. Dobb's Journal of Software Tools* 1994; **19**(4):38,40,98,99.
72. Microsoft. The security support provider interface. *White paper*, Microsoft Corporation, Redmond, WA, 2000.
73. Myers J. RFC-2222: Simple Authentication and Security Layer (SASL). *Network Working Group Request for Comments*, October 1997.
74. Miller BP, Christodorescu M, Iverson R, Kosar T, Mirgordskii A, Popovici F. Playing inside the black box: Using dynamic instrumentation to create security holes. *Parallel Processing Letters* 2001; **11**(2/3).
75. Thain D, Livny M. Bypass: A tool for building split execution systems. *Proceedings of the 9th IEEE Symposium on High Performance Distributed Computing (HPDC9)*, Pittsburgh, PA, August 2000. IEEE Press: Piscataway, NJ, 2000; 79–85.
76. Anstreicher K, Brixius N, Goux J-P, Linderoth J. Solving large quadratic assignment problems on computational Grids. *Mathematical Programming*. Springer: Berlin, 2000.
77. HawkEye. http://www.cs.wisc.edu/condor/hawkeye [August 2002].
78. Public Key Infrastructure Lab (PKI-Lab). http://www.cs.wisc.edu/pkilab [August 2002].
79. NSF Middleware Initiative (NMI). http://www.nsf-middleware.org [August 2002].

