19

CHAPTER

Building Reliable Clients and Services

Douglas Thain and Miron Livny

Traditional distributed computing is dominated by a client–server model. File systems, databases, and hypertext systems are all designed with the assumption that a few powerful and reliable servers support a large and dynamic set of clients. Servers are complex and expensive, while clients are lightweight, cheap, and simple. Servers are responsible for persistence, security, and coherence, while clients are responsible for very little. Client software often acts directly on behalf of an interactive user who steers the overall interaction.

Grid computing is different: a single Grid client may harness large numbers of servers over extended time periods. For example, one client may consume more than 100,000 CPU-hours in one week on systems distributed worldwide (443) (see also Chapter 10). As a consequence, both Grid clients and servers are multiplexed, multiprotocol, and multithreaded. This organization could be termed *peer-to-peer* (Chapter 29) to indicate that participants are equals, although it need not be the case that participating systems are poorly connected, particularly unreliable, or mutually untrusting, as is often assumed in peer-to-peer computing. The key point is that multiple parties—both consumers (clients) and providers (servers) of capabilities—must act in concert to achieve an overall goal in a reliable fashion. Each must meet obligations relating to security, performance, and progress. Thus, each requires techniques to overcome various kinds of failure conditions, and to track long-lived interactions that may stretch over days or weeks.

Preceding chapters have introduced the basic principles of service-oriented architecture and OGSA (279), explained why future Grid services will have persistent state and active and complex responsibilities (Chapter 17), and introduced the notion of a service-level agreement (SLA) as a means of negotiating expectations

concerning the duties that other parties will fulfill (Chapter 18). In this chapter, we introduce *client-oriented architectures* that may be used to achieve reliable distributed execution in complex, distributed, and dynamic Grid environments. Building on experiences within the Condor Project—which has specialized in the problem of reliably executing jobs on remote machines for almost two decades—we present specific techniques for use in three different contexts: remote execution, work organization, and data output.

19.1 PRINCIPLES OF RELIABLE SYSTEMS

We first introduce some general design principles that apply to distributed systems and, in particular, to the complex applications often found in Grid computing. We shall encounter applications of each of these principles in our discussion of remote execution, work management, and data output.

Effective operation requires responsible behavior. Responsibility is a common notion in distributed computing. Random behavior is rarely acceptable; rather all parties are required to operate within certain limits of decorum. Here are some well-known standards of conduct:

- The Ethernet (474) discipline arbitrates access to a shared medium without a central controller. The discipline expressly prohibits transmission at will. Civilized clients must listen for a quiet interval before broadcasting, and then double-check to make sure that transmissions were broadcast correctly. If an accidental collision occurs, a client has the responsibility to inform others that may be affected by using a short attention burst, and then must fall silent again for a specified time. If Ethernet clients simply tried to "shout" louder and faster to drown out everyone else, the medium would be entirely unusable.
- The two-phase commit protocol (327) allows for the atomic acceptance of a series of nonatomic operations. This protocol places strong burdens on both coordinators (clients) and servers. During the assembly of a transaction, either side may abort at will. However, once a server accepts a coordinator's request to prepare the transaction, it may not release it of its own accord. Conversely, the coordinator has the obligation to try forever until it successfully completes a prepared transaction. An abdication of responsibility from either side would lead to an inconsistent system and potentially the loss of data.

- The Jini (668) resource discovery system allows a client to obtain a lease on a server. Once allocated, the server accepts the responsibility of serving only the lessee. Likewise, the client accepts the responsibility of periodically renewing the lease. If a communication failure prevents renewal, both sides have the obligation to assume the lease is broken, thus preventing waste and inconsistent states.

The client is ultimately responsible for reliability. This is a restatement of the end-to-end design principle (570). In the final reckoning, the consumer is responsible for his/her own well-being. The careful grocery shopper always checks a carton of eggs for cracked shells. So too must the Grid client verify that job outputs are coherent and complete.

Reliable services are difficult to manage. This theme is counterintuitive. It would seem that services that make greater efforts to be resilient to failures would be more useful and reliable. Yet, the opposite is frequently true. More reliable services often have more complex and unexpected failure modes. Sometimes the best service is the one that fails frequently in a predictable way.

Soft state simplifies garbage collection. As discussed in Chapter 17, soft-state mechanisms (e.g., leases (323)) can simplify the recovery of resources following abnormal (as well as normal) task termination.

Logging simplifies persistent state. Grid services that manipulate persistent state must remember completed actions, track requests in the process of assembly, and deal with requests that were started but perhaps forgotten. A log—a stream of event records that survives crashes—is a standard tool for this job. Using well-known techniques, the history and current state of a service may be easily recovered from the log.

19.2 RELIABLE REMOTE EXECUTION

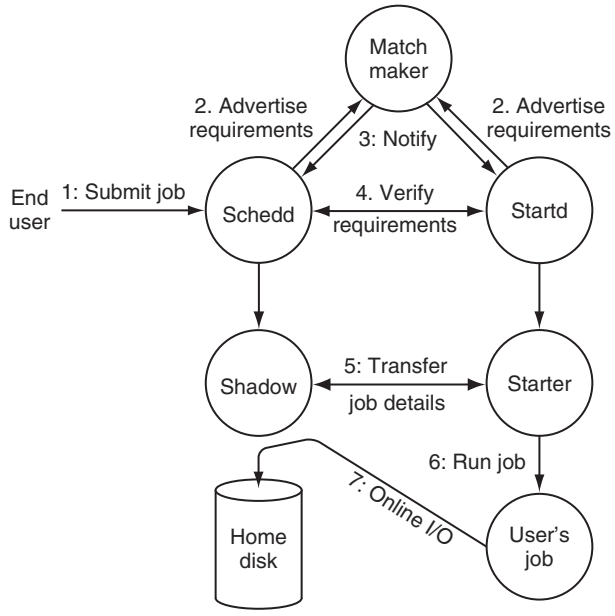
We next turn to the deceptively simple problem of reliable remote execution. We illustrate our discussion with two examples of large-scale distributed system—traditional Condor and Grid-enabled Condor-G—that achieve reliable remote execution in different ways. Traditional Condor, designed to deal with opportunistic resources, focuses on the problems of ensuring that failures are detected rapidly and forcing a quick reset to a known state. However, the techniques used to

achieve this goal are less well suited for use in a wide-area network in which remote servers are persistent. Condor-G allows for network disconnections while a job is managed by a remote batch system, but this introduces new and complex challenges for system coherency.

19.2.1 Condor

The core components of the traditional Condor distributed batch system (Figure 19.1) form the Condor *kernel*. Each component must discharge a certain responsibility, as we now describe:

The *schedd* is a reliable client of Grid computing services. It serves as the entry point for end users, providing a transaction interface for submitting,



19.1 FIGURE

The Condor kernel: the seven steps to run a Condor job. (1) The user submits the job to a schedd. (2) The schedd and the startd advertise themselves to a matchmaker. (3) The matchmaker notifies two compatible parties. (4) Both parties verify that they match each other. (5) The shadow and starter are created, and communicates the details of the job to be run. (6) The starter executes the user's job. (7) If needed, the job performs I/O by communicating with the shadow directly or through a proxy in the starter.

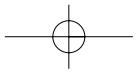
querying, and removing jobs. It is also responsible for persistently storing a user's jobs while finding places for them to execute. The schedd is responsible for enforcing user requirements on job execution. For example, the user may require that jobs only run on machines that have sufficient memory and that are owned by a trusted user.

The *startd* is a reliable Grid computing service. It manages an execution machine, and is responsible for finding work to be done within the constraints placed on it by the machine's owner. For example, an owner might permit only jobs submitted by the "physics" group to be run during the day while allowing any sort of job to run at night. Jobs that satisfy these requirements may also be ranked. The same owner might prefer, but not require, jobs from the "chemistry" group at night. When supervising a job, the startd is responsible for monitoring the state of the machine, possibly evicting and cleaning up after a job if the owner's requirements are no longer met.

The *matchmaker* is responsible for introducing potentially compatible consumers (schedds) and producers (startds). The matchmaker accepts *advertisements* from all parties, written in the ClassAd (538, 539, 541) resource description language, describing their current state and the properties of the resources that they seek. Once a potential match is found, both resources are notified. The matchmaker is also responsible for enforcing system-wide policies that cannot be enforced by a single schedd or startd. For example, the matchmaker controls admission to the pool as well as the fraction of machines allocable to any given user. It also performs accounting to track pool-wide resource use.

Although the matchmaker informs compatible parties that they are a potential match, each party still bears a responsibility to enforce its own requirements and to translate the potential of a match into productive work. The match could be bad if it is based on stale information or if the matchmaker is untrustworthy. Thus, schedds and startds contact each other directly to verify that their requirements are still met before attempting to execute a job. Once satisfied, they may begin to work together. However, either side may abort the match at any time if it discovers that its requirements are no longer satisfied.

Two subprocesses—a shadow and a starter—work together to actually execute a job. The *starter* is responsible for creating an execution environment at the remote site. Like a command-line shell, it transforms a job description into the operating system calls that actually execute it. It creates a working directory, sets up standard I/O streams, and monitors the job for its exit status. Most importantly, it is responsible for informing the other Condor components whether the job was able to execute in the available environment. Although the starter provides all mechanisms needed to execute a job, it does not provide policies. The starter relies entirely on the shadow to decide what to run and how to do it.



The *shadow* is responsible for making all policy decisions needed by a job. Upon request, it names the executable, arguments, environment, standard I/O streams, and everything else necessary for a complete job specification. When the job terminates, the shadow examines the exit code, the output data, the execution machine, and any other relevant information to determine whether the job has truly run to completion, or has simply failed in the current environment. The shadow also serves as a basic data server for the job. It provides a remote I/O channel that the user's job may call either directly or via a proxy in the starter. This channel may be used to fetch the executable from the home site and to perform online input and output. (Note that the data server role may be filled more efficiently by third parties such as storage appliances (637) and checkpoint servers (101). The primary responsibility of the shadow is to control how and when these resources are used.)

We categorize these processes according to their reaction to failures. *Resilient processes* are designed to handle a wide variety of error conditions via standard techniques such as retry, logging, resetting, and notification. *Brittle processes* simply abort and exit upon detecting any sort of unexpected condition. As might be expected, resilient processes can be complex to develop and debug, generally requiring the management of persistent state and logging in stable storage. Brittle processes are easier to create and maintain.

The shadow and starter are brittle processes. Throughout the execution of a job, they maintain a TCP connection with each other that is used for communicating both the job's remote I/O and execution details such as the exit code and resource use. If either side should detect that the connection is broken, it immediately aborts, killing itself entirely. If the local system is not prepared to execute a job, whether it is due to a lack of memory, an offline file system, a corrupted operating system, or even just a bug in Condor, the same form of abort is used, thus breaking the TCP connection and forcing the peer to take the same action.

Although these processes are brittle, they may still manipulate persistent state. For example, both make use of local logs to record their progress. The shadow even manipulates the job-state database stored in the schedd to indicate the disposition of the job and the amount of resources it has consumed. If either the shadow or starter should fail unexpectedly, the job state does not change. Thus, the mere disappearance of a job cannot be considered to be evidence of its success.

Neither the shadow nor the starter is responsible for cleanup. This task is left to the resilient schedd and startd, which enforce the lease that was negotiated to use the machine. Each process monitors the state of the machine on which it runs as well as the disposition of its shadow and starter children. If any child process should exit, then the resilient processes are responsible for cleaning up by killing

runaway jobs, deleting temporary disk space, and discarding unused credentials. Unlike Condor-G, which we describe in Section 19.2.2, no remnant of the job is left behind after a failure.

This enforced cleanup ensures the consistency of the entire system. The startd must not simply restart a job after a failure, because the schedd will not necessarily know (or desire) that the job continue there. Rather, the schedd is left free to choose the next site of the job. It might try again at the location of the last failure, it might attempt to use another known but idle machine, or it may begin matchmaking all over again. The schedd is charged with this responsibility precisely because it alone knows the state of the job.

The matchmaker is also a brittle process, although it has few error conditions to encounter. If the matchmaker process or machine should crash, running schedds and startds will not be affected, although no further matches will be made. Once restarted, a matchmaker will quickly rebuild its state from the periodic updates of startds and schedds, and the pool will operate as before.

Every machine that runs some component of Condor is managed by a resilient *master* process. This process has no duty except to ensure that schedds, startds, and matchmakers are always running. If such a service process should terminate, whether due to a bug, administrative action, or just plain malice, the master logs the event, restarts the process, and may inform the human system administrator. Repeated failures are interspersed with an exponentially increasing delay to prevent busy-looping on unexpected conditions. This ability to start and stop processes is also used as a remote "master switch" to enable and disable Condor simultaneously on large numbers of machines. The UNIX "init" process manages the master process and thus handles system startup, shutdown, and errors in the master itself.

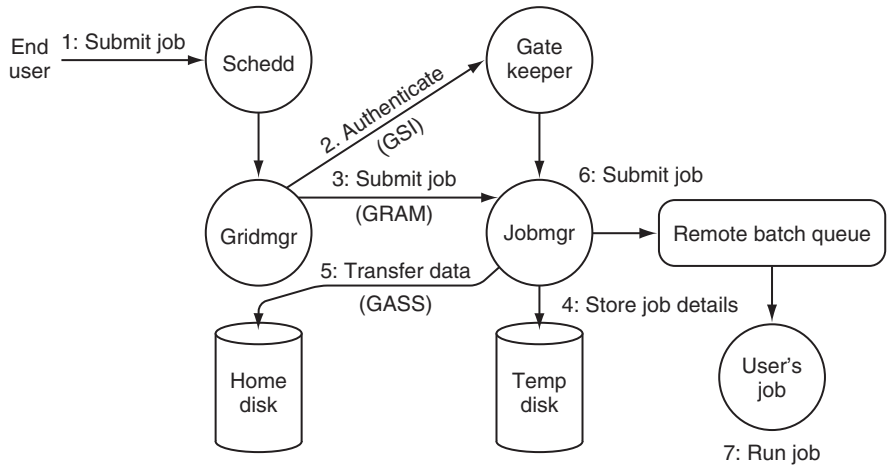
This dichotomy of resilient and brittle processes has several advantages. As an engineering model, it drastically simplifies the addition of new technologies and features to serve running jobs. The vast majority of these are placed in the brittle processes, where error conditions and even faulty code are cleanly handled by the controlled destruction of the whole process. The resilient processes are naturally more complicated, but only take part in the fundamental matchmaking interaction, which is well debugged and unaffected by the addition of most new features.

However, the brittle interaction between the shadow and the starter has one significant drawback. There must be a reliable network connection between the submission and execution sites for the entire lifetime of a job. If it is broken, the job is not lost, but a significant amount of work must be repeated. To permit temporary disconnections between the two sites requires that both sides become resilient processes with persistent state and a more complex interaction. Exactly this model is explored in Condor-G.

19.2.2 Condor-G

As discussed in Chapter 4, Grid computing technologies developed within the Globus Project (276) provide standard protocols and services for accessing remote resources. In particular, the Grid Security Infrastructure (GSI) (280), Grid Resource Allocation Manager (GRAM) (205), and Global Access to Secondary Storage (GASS) (117) protocols can be used to construct secure remote execution systems that cross institutional boundaries while making use of existing batch systems without modification. However, the successful orchestration of these three protocols is no small matter, requiring a client that can persistently manage a large number of jobs even in the face of complex failures. To fulfil this role, the Condor and Globus projects collaborated to build Condor-G (291), a Grid-enabled agent for accessing remote batch systems.

In the Condor-G architecture (Figure 19.2), as in traditional Condor, the end user interacts primarily with a *schedd*, which keeps all job state in persistent storage. However, rather than specify the general requirements each job has for an execution machine, the user must specify the specific name of a Globus GRAM server that represents an entire remote batch system.



19.2 FIGURE

Condor-G architecture—the seven steps to run a Condor-G job. (1) The user submits a job to the schedd, which creates a gridmanager for the job. (2) The Gridmanager authenticates the user to the gatekeeper, which creates a jobmanager. (3) The Gridmanager transmits the job details to the jobmanager. (4) The jobmanager stores the details. (5) The jobmanager transfers the executables and input data from the home site. (6) The jobmanager submits the job to the remote batch queue. (7) The job executes.

For each job submitted, the schedd creates a process called a *gridmanager*, which is responsible for contacting a remote batch queue and providing the details of the job to be run. The gridmanager is analogous to the shadow in traditional Condor. It is responsible for actively seeking out a remote batch system through its main point of contact, the gatekeeper. Once a job begins running, the gridmanager is also responsible for servicing the job's input and output needs through the GASS protocol.

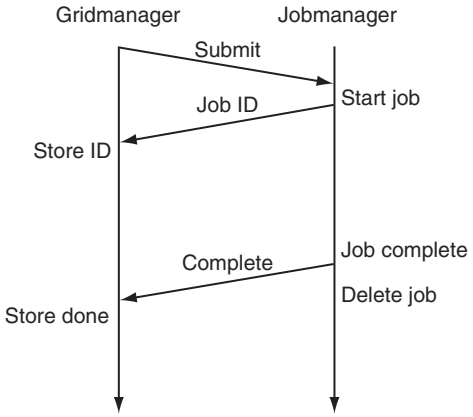
The *gatekeeper* is responsible for enforcing the admission policies of a remote batch queue. Using GSI, it accepts connections, authenticates the remote user, and maps the remote credentials into a local user ID. Once authenticated, the client may request a particular service, such as access to a batch queue through a jobmanager. The gatekeeper creates the new service process with the local user ID and passes the connection to the remote user.

The *jobmanager* is responsible for creating a remote execution environment on top of an existing batch execution system. A variety of jobmanagers is available for communicating with batch systems such as LoadLeveler (16) (a descendant of Condor), LSF (702), Maui (385), and PBS (31). A jobmanager could even pass jobs along to a traditional Condor pool like that discussed in the previous section. The jobmanager bears a responsibility similar to that of the traditional Condor starter, but it differs in one significant way. Unlike the starter, it uses the local batch system to hold the details of a job in persistent storage and attempts to execute it even while disconnected from the gridmanager. This strategy permits a system to be more resilient with respect to an unreliable network, but also creates significant complications in protocol design.

We use the GRAM protocol design to illustrate several important issues relating to remote interactions. An early version of the GRAM protocol design, shown in Figure 19.3, used atomic interactions for the submission and completion of jobs. For example, the submission of a job from the gridmanager to the jobmanager consisted of a single message containing the job details. Upon receipt, the jobmanager would create a unique name for the job, store the details under that name, and return the ID to the gridmanager. Thereafter, it would transfer the executable and input files and attempt to submit the job to the remote batch queue.

However, a failure at any of several key points in this interaction would cause trouble. Some crashes would result in an *orphan* job, left running and consuming CPU and storage, but with no way to stop or recover it. Other crashes would result in a *wasted* job, which ran successfully, but could not communicate its results back to the submitter. For example:

- An orphan would be created if the network connection failed after the jobmanager stored the job details but before it could return the job ID. In this



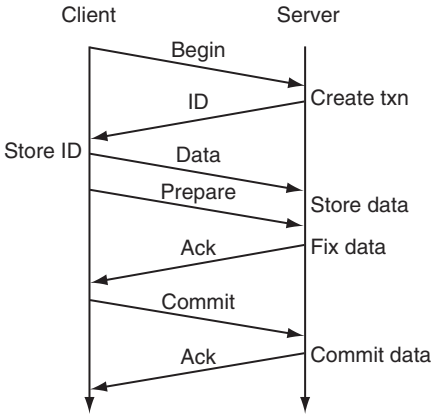
19.3 FIGURE

The original GRAM protocol used between the gridmanager and jobmanager. It consisted of two atomic interactions: one to submit a job and one to indicate job completion. This protocol can lead to orphaned or wasted jobs as described in the text. A standard solution to this problem is shown in Figure 19.4.

case, the job would run in the batch system, the jobmanager would oversee it, but the gridmanager would not know its ID.

- An orphan would also be created if the machine running the gridmanager crashed after the job ID was returned. In this case, the job would run in the batch system, and the gridmanager would know its ID, but there would be no jobmanager to oversee it.
- A job would be wasted if any crash or network failure prevented the job completion message sent by the jobmanager from reaching stable storage at the submitter. In this case, the jobmanager would delete the completion details immediately after sending the message, leaving no possibility for the gridmanager to request them again after a failure.

These problems can be solved by the use of *transactions* for grouping and naming commands and the *two-phase commit* protocol for an orderly completion. A standard two-phase commit protocol is shown in Figure 19.4. Here, the transaction is created by a *begin* message, which causes the server to create a new transaction, naming it with a unique ID that must be logged and is returned to the client, which does the same. The ID is used to identify the transaction in all further messages. One or more following *data* messages fill the named transaction with all of the details the client wishes to provide. A *prepare* message ends the first phase, causing the server to check the transaction's validity and fix it in permanent storage,



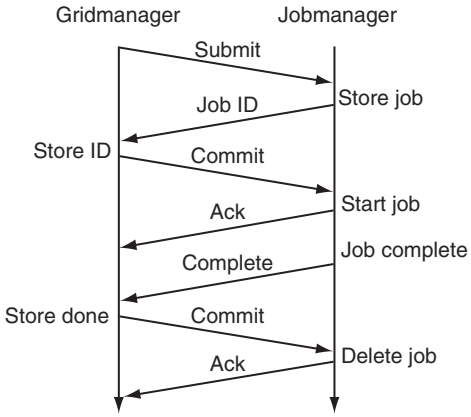
19.4 FIGURE

A standard two-phase commit protocol between a generic client and server. The first phase consists of a begin message to create a transaction, one or more data messages to provide the transaction details, and a prepare message to fix the transaction in persistent storage. The second phase is consummated with a commit message. Condor-G uses a variation of this protocol shown in Figure 19.5.

while stopping short of actually completing the operation. In the second phase, a *commit* message forces the prepared transaction to run to completion and frees the log. An acknowledgment from the server to the client permits the client to free its own log. At any time, an *abort* message (not shown) may be used to destroy the transaction completely.

The two-phase commit protocol allows for easy recovery from a failure of either party or the communication medium. If any message following begin is lost, the client may resend it without harm, using the logged transaction ID to refer to the correct operation. The separation of prepare from commit allows a single client to initiate multiple transactions at multiple servers, committing all of them only if prepare is accepted at all services. The worst loss that can happen with this protocol occurs if the response to begin is lost. Because the transaction ID is unknown to the client, it must start a new transaction, thus wasting a small amount of storage at the server. Such incomplete transactions may be removed periodically by a garbage collector.

These considerations led to a redesign of the GRAM protocol in Globus Toolkit version 2 (GT2). As the introduction of a complete two-phase commit protocol would have involved a costly rewrite of a large portion of the protocol, this redesign reaped most of the benefits by employing a limited form of two-phase commit through small changes to the submission and completion elements of the protocol. These improvements are shown in Figure 19.5.



19.5 FIGURE The improved GRAM protocol, which uses a limited form of the two-phase commit protocol given in Figure 19.4. In the first phase, the gridmanager submits the job details to the jobmanager. In the second phase, the gridmanager issues a commit to complete the transaction. A similar technique is used to indicate job completion.

To submit a job, the gridmanager issues a submit message as before. However, this message is interpreted as a combination of begin, data, and prepare. It requests a new transaction, specifies all of the job details, and forces the data into persistent storage. The jobmanager then responds with a unique job ID. To complete the transaction, the gridmanager issues a commit message that is acknowledged by the jobmanager. A similar convention is used in reverse when the job is complete. The jobmanager sends a complete message to the gridmanager, informing it that the job has completed. The gridmanager responds with another commit message, indicating that it is safe to delete the entire transaction, and the jobmanager acknowledges.

This protocol provides coherence in the event of failures. After the initial submit message, either party or the network may fail, but the protocol may resume by referring to the existing transaction ID. The greatest loss could occur if the submit message is successful but its reply is lost. However, this amount of data is relatively small. Large binary objects such as the executable and input files are transferred asynchronously by the job manager only after the submission has been committed.

Another challenge in the early GRAM design lay in the fact that the jobmanager stored locally persistent state and dealt with two other persistent services: the gridmanager and the batch queue. Yet the jobmanager itself is not persistent. If the entire system should lose power and reset, then the gridmanager and batch queue

would recover and resume, but there would be no jobmanager to oversee the running job. This is a common and dangerous situation: the schedd is aware that the job is in a remote batch queue, but without the jobmanager, it cannot control it.

To remedy this situation, GRAM in GT2 and later Globus Toolkit releases is equipped with a restart capability. If the schedd loses its connection with the jobmanager, it restarts another one by contacting the gatekeeper and requesting another jobmanager of the same type. It then asks the newly minted jobmanager to recover the persistent state of job in question and to continue monitoring the running job. Thus, so long as the schedd itself recovers, it can assume responsibility for starting the jobmanager.

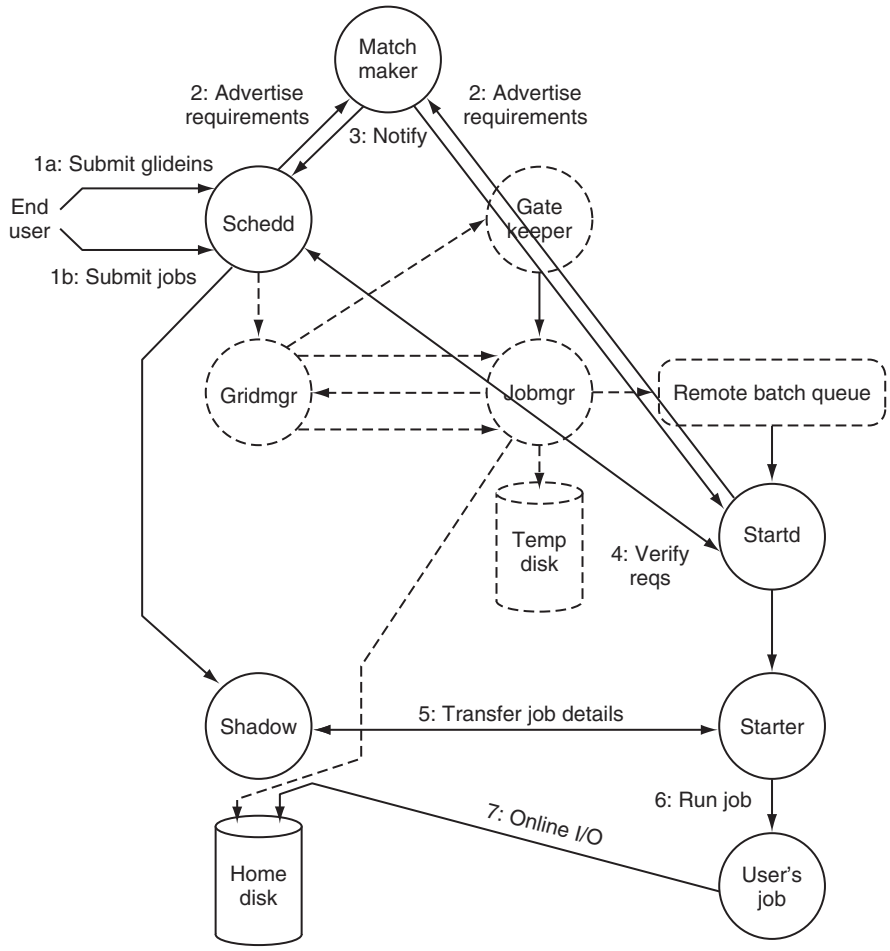
It should be noted that this new form of resilience places an important obligation on the jobmanager. Because the schedd creates a new jobmanager whenever needed, any jobmanager has the obligation to destroy itself whenever contact with the schedd is lost: otherwise, it would be possible for two jobmanagers to oversee the same job. In the same way that the shadow and starter are obliged to be brittle in traditional Condor, the jobmanager is also obliged to be brittle because another process—the schedd—is responsible for its resilience.

This discussion emphasizes the point that resilience introduces complexity. The brittle properties of the traditional Condor system make failure recovery simple. However, the resilient nature of every process in Condor-G makes failure recovery much more complicated. The standard techniques established by highly consistent systems such as distributed databases must not be overlooked when designing computational Grids.

19.2.3 Gliding-In

Both traditional Condor and Condor-G have distinct advantages and disadvantages. Traditional Condor allows for powerful selection and description of resources and provides specialized environments for checkpointing, remote I/O, and the use of runtime systems such as PVM, MPI, and Java. Condor-G allows a user to reliably harness any remote computation system, even if it is not a Condor pool. The two approaches may be combined to leverage the advantages of both via a strategy called *gliding-in* that builds a traditional Condor pool on top of a Condor-G system.

The glide-in technique is shown in Figure 19.6. First, the Condor software is packaged into a "glide-in job" that is given to Condor-G. Next, the user estimates approximately how many machines they wish to use, and submits that many glide-in jobs. Once running under the remote batch queue, these processes form an ad hoc Condor pool with an existing public matchmaker or perhaps a private



19.6 FIGURE

Gliding in Condor via Condor-G. (1a) A user submits a glide-in job to a Condor-G schedd. They are transferred to a remote batch queue as described in Figure 19.2. (1b) A user submits a normal job to the same schedd. (2–7) Jobs execute on the glided-in Condor system as in Figure 19.1.

matchmaker established by the user. Of course, the size of the pool depends on how many glide-in jobs propagate through the remote queue and actually begin running: the user only exercises indirect control through the number of glide-in jobs submitted. Once the ad hoc pool is formed, the same user may then submit jobs of actual work to be done. These jobs perform matchmaking and execute on the glided-in resources just as in traditional Condor.

The glided-in Condor processes make the remote batch queue a more friendly execution environment by providing the specialized Condor tools and language environments. They also allow the user to perform resource management without involving the host batch system; jobs may be preempted, migrated, or replaced at the user's whim without involving the underlying system. The glided-in Condor pool also offers the user some insurance against runaway consumption. If some oversight in the protocol between the gridmanager and jobmanager should accidentally leave a job running in the remote batch queue, it will appear in the roster of the ad hoc Condor pool where the user can either harness it or deliberately remove it. If the glide-in jobs themselves are left behind and unused, they will automatically terminate themselves after sitting idle for a user-defined length of time. Thus, the user need not worry about misplaced jobs.

19.3 WORK MANAGEMENT

So far, we have described a user's needs as simply a *job:* a single process to be executed on one machine. (Of course, a single job might be many tasks destined to run on a multiprocessor. However, such a job and machine would each be represented as one entity within a Grid.) However, most users come to the Grid to accomplish a large amount of work, frequently broken into small independent tasks. They may be performing parameter sweeps, rendering digital video, or simulating a complex system. The performance of such workloads is not measured in traditional microscopic computing metrics such as floating-point operations per second, but rather in end-to-end productive terms such as permutations examined per week, video frames rendered per month, or simulations completed per year: what is known as *high-throughput computing* (539) (see also Chapter I:13).

Users need higher-level software to manage such large-scale activities. Systems such as Condor and Condor-G manage work that is ready to be performed. Users also need to track work that is yet to be done, has already completed, or perhaps needs to be repeated. In addition, they need structures that control the order, priority, and assignment of work to Grid resources. The Condor Project and its collaborators are investigating a number of these structures, which may be divided into two categories: job-parallel and task-parallel.

A *job-parallel* system considers a body of jobs to be done, chooses the next jobs to execute, and then looks for resources on which to execute them. Job-parallel systems are well suited to workloads where each job requires distinct resources. The primary tool for job-parallel computing in the Condor Project is the directed acyclic graph manager, or DAGMan for short. DAGMan is used in

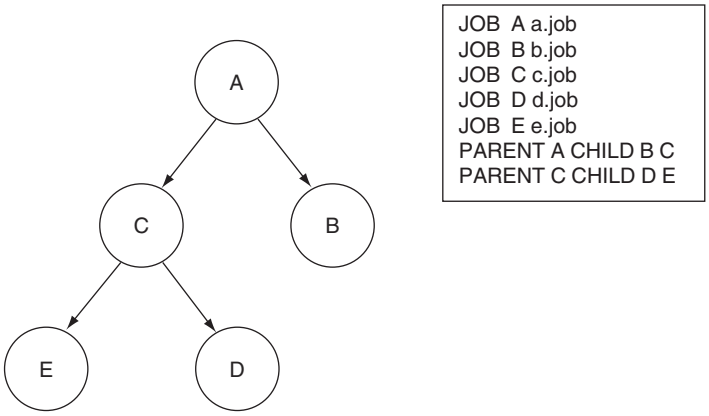
several settings, including the Condor and Condor-G systems described previously. With slight variations, it is used in two other Grid systems that we describe below.

A *task-parallel* system performs a similar task in the opposite order. It considers the (potentially changing) resources it has currently available, and then assigns a task from a work list to a suitable resource. Task-parallel systems are well suited to workloads in which tasks are small and have similar requirements. We describe a task-parallel framework known as master–worker that has been used in several production settings, most notably to solve a significant optimization problem. A similar task-parallel model is used by the XtremWeb system, which works in concert with Condor.

19.3.1 Job-Parallel Computing

Figure 19.7 shows a *directed acyclic graph*, or DAG for short. A DAG consists of several nodes, each representing a complete job suitable for submitting to an execution system such as Condor or Condor-G. The nodes are connected by directed edges, indicating a dependency in the graph. For example, in Figure 19.7, job A must run to completion before either job B or C may start. After A completes, jobs B and C may run in any order, perhaps simultaneously.

DAGMan is the Condor process responsible for executing a dag. DAGMan may be thought of as a distributed, fault-tolerant version of the standard UNIX



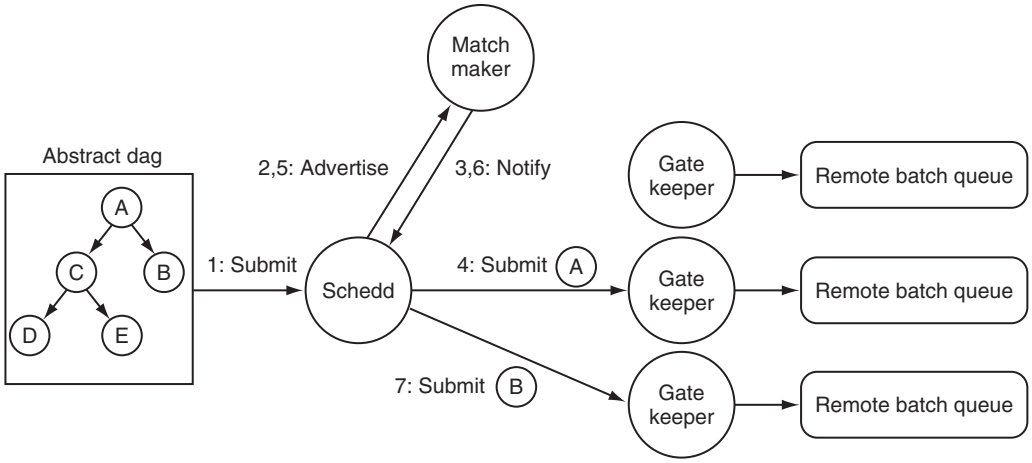
19.7 FIGURE A directed acyclic graph of jobs or DAG.

make facility. Like make, it ensures that all components of a DAG run to completion in an acceptable order. Unlike make, it does not rely on the file system to determine which jobs have completed. Instead it keeps a log to record where jobs are submitted, when they begin executing, and when they complete. If DAGMan should crash, it can replay the log to determine the progress of the DAG.

DAGMan executes jobs by submitting them to a schedd in a Condor or Condor-G pool. As described earlier, the schedd ensures that the jobs are persistently retried, even if system components (including schedds) crash. To provide DAGMan with the same sort of resilience, DAGMan itself is submitted to the schedd, which considers it to be a normal Condor job in every respect, except that it executes under the supervision of the schedd on the submitting machine. Thus, the same transactional interface can be used to submit whole DAGs, inquire about their status, and remove them from the queue

Each job in a DAG carries its own independent requirements on execution. In a traditional Condor pool, one job in a DAG may require a machine with a large amount of memory, while another may require a machine with a particular type of CPU. In a Condor-G system, different jobs may need to run on entirely different gatekeepers. Each job maintains its own abstract requirements that are not evaluated until the job has been passed to the schedd for consideration by the matchmaker. We call this property *late binding*. The job is not assigned an execution site until the last possible moment. Thus, the system has maximum flexibility in deciding how to execute the job. Other systems have explored the possibility of binding earlier in the lifetime of a job. We discuss two such systems here, the EU Data Grid and the Chimera system.

The planned EU Data Grid (EDG) is a Globus Toolkit-based Grid that links computing centers and data storage centers across Europe to support the computing needs of a variety of data-intensive science projects. In the planned EDG architecture (Figure 19.8), work is expressed as an abstract DAG consisting of jobs with requirements on the site of execution. In addition to the usual requirements such as the necessary CPU, amount of memory, and so forth, these jobs may also have requirements on relationships between elements. For example, one job may need to execute on a machine with a certain dataset in local storage, while another job may simply need to run at the same site as its predecessor in the DAG. Armed with this information, the schedd evaluates the DAG piece by piece. For each job, it employs the traditional model of consulting with a matchmaker to find an appropriate match for the job. However, instead of matching with a single machine, it matches with an entire batch queue. It then uses the Condor-G mechanisms for executing the job through that queue. This process continues until the entire DAG has been evaluated and executed.



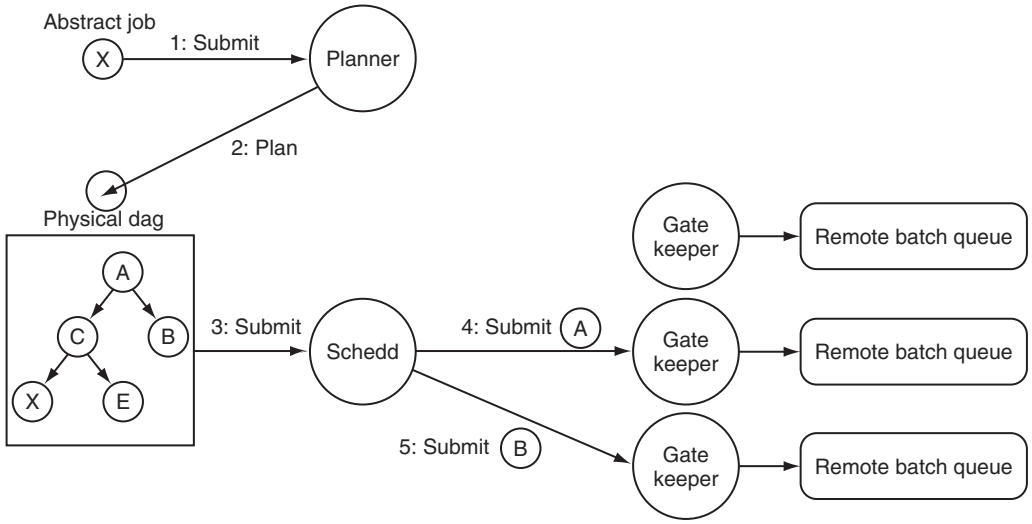
19.8 FIGURE

The planned architecture for executing dags in the European Data Grid. (1) The user submits a DAG composed of jobs with abstract requirements. (2) The schedd advertises the requirements of the first job. (3) The matchmaker notifies the schedd of a compatible gatekeeper. (4) The schedd executes the first job via Condor-G as in Figure 19.2. (5) The schedd advertises the second job. (6) The matchmaker notifies the schedd. (7) The schedd executes the second job. This pattern continues until the DAG is complete.

The Chimera system (284, 285) is designed for the production of *virtual data*. Unlike a traditional batch execution system, Chimera does not require the user to specify what programs are to be executed. Rather, it asks the user to specify what data are to be produced. Much like a functional programming environment, Chimera plans what procedures are necessary to realize the data using fundamental inputs. Thus, the user is relieved of the burden of tracking and managing large numbers of batch jobs, while the system can silently optimize caching, data placement, and function shipping. The architecture of the Chimera system is shown in Figure 19.9.

The user interacts with the Chimera system by submitting request for specific virtual data elements. Chimera consults a *virtual data catalog* to determine whether the data have already been created, and may potentially respond immediately with the result if it is available. Otherwise, Chimera produces a DAG designed to compute the necessary data. This DAG is executed via the use of many of the Condor-G components described previously.

The Chimera architecture admits to a variety of scheduling strategies, from early to late binding. The Pegasus planner (216) performs early binding, using



19.9 FIGURE

The architecture of the Globus Chimera virtual data system. (1) The user submits an abstract job that describes a data object to be realized. (2) The planner creates a DAG that will realize the data. Each job in the DAG is bound to a physical location. (3) The DAG is submitted to a schedd. (4) Each job in the DAG is executed via Condor-G as in Figure 19.2.

various planning techniques to produce a concrete DAG composed of jobs whose location is already specified in the form of a GRAM gatekeeper name. The whole DAG is passed to a schedd, which then executes each component via Condor-G as in Figure 19.2. Other approaches (544) that perform late binding via call-outs from the Condor-G agent to simple decision makers are being explored.

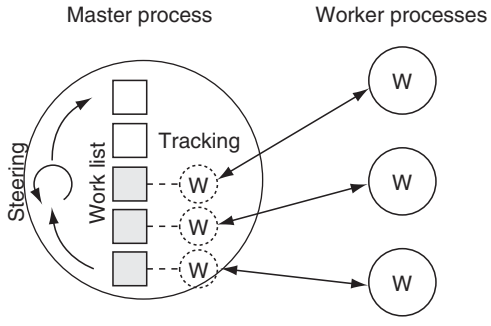
All of these examples use DAGMan to manage job-parallel workloads, but they differ in how and when abstract names are assigned to physical resources. The traditional Condor approach of *late binding* defers this decision until the last possible moment. If the job should fail, an entirely new name binding may be attempted and the job sent elsewhere. In the EDG *medium binding* approach, the user still specifies abstract requirements on when and where each component must be executed, but the schedd makes a single decision regarding which batch queue to use. Once made, the final assignment of compute resource is left to the local batch scheduler. Finally, in the Chimera–Pegasus *early binding* approach the user is to specify an abstract target of the work, but the planner decides on the entire set of physical resources to use before the work is even submitted to Condor-G. The schedd simply passes the jobs to the named resources.

None of these approaches is universally better than the others. Early name binding is needed when the information needed to make a good placement decision is not integral to the remote execution system. This is appropriate in Chimera because the planner has information about storage resources that is not available to Condor. Late name binding is better when such information is available, because the system is able to make alternate choices when failures and other exigencies thwart the original plan. An example of this is found in the notion of *I/O communities* (637) that integrate the state of storage into an existing matchmaking system. Medium name binding is a compromise that works when storage resources are accessible to the matchmaker, but not to the target execution system.

19.3.2 Task-Parallel Computing

The task-parallel model works by first harnessing the necessary workers and then assigning work units as workers are available to process them. This computing model is naturally suited to large problems of indeterminate size that may be easily partitioned. We discuss two task-parallel computing systems: Condor master–worker (Condor-MW) and XtremWeb (XW); see also Chapters 28 and 29.

In the Condor-MW and XW systems (Figure 19.10), as in other similar systems such as Entropia (180), SETI@home (79), and Nimrod (56), one master process directs the computation with the assistance of as many remote worker processes as it needs or the computing environment can provide. The master itself contains three components: a work list, a tracking module, and a steering module. The



19.10 FIGURE The master–worker framework.

work list simply records all outstanding work. The *tracking module* accounts for the remote worker processes and the remaining work to be done and assigns workers uncompleted work. The *steering module* directs the computation by assigning work, examining results, and modifying the work list.

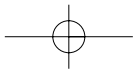
Task-parallel computing is naturally fault-tolerant. Workers are assumed to be unreliable: they disappear when machines crash and they reappear as new machines become available. If a worker should fail while holding a work unit, the steering module simply returns it to the work list. The steering module 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 near the end of a problem.

Workers themselves must be managed. New workers must be sought out when needed, failed workers must be noted and replaced, and idle workers must be released when no longer needed. These functions can be performed within the system itself or by the user. We explore both possibilities.

The Condor-MW (443) system is a set of C++ classes that encapsulate communication and management facilities needed for task-parallel computing. The programmer extends these classes to construct an application-specific master and worker. The management of workers may be tightly integrated with the work itself. For example, as a computation progresses, the master may want to scale the number of workers with the size of the work list. This capability is supported by the function set_target_num_workers, which the application writer can call at any point in a program (534) to request that the master attempt to adjust the number of workers, using the underlying Condor schedd to request new resources or release old ones. Of course, the number of workers is a goal, not an assertion. The number of workers may be less than the target if there are no more available machines, and may be higher if the master slowly releases workers as they finish their current tasks.

Communication between master and workers is achieved via TCP connections, thus allowing the master to know when a stream has broken, indicating that a worker has disconnected or failed. A distributed file system may also be used as the communication channel, in which case messages between master and workers are placed in a rendezvous directory that serves as a persistent message queue. This approach allows MW to tolerate network outages, relying on the underlying file system for reliable delivery. In this model, timeouts must be used to detect the failure of a worker process. A third option is to use PVM communications.

The master dispatches work units to workers and thus maintains tight control over the disposition of each worker. It knows exactly which are busy and which are idle, and can even deliberately hold workers idle in the expectation of future work or as an instantaneous reserve against failures. (In Condor, a client is "charged" for machines that it claims, whether it uses them or not. Typically, the



charge is only accumulated as a convenience figure to measure pool use and control priority, but it could also be turned into a pecuniary debt.) Because the master controls the workers closely, it may distinguish between failed and idle workers and seek out new resources as needed.

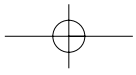
The XW (257) system is a Java-based system for task-parallel computing. Like Condor-MW, it encapsulates many of the communication and organizational details for task-parallel computing, allowing the programmer to concentrate on application details. XW differs from Condor-MW in several key ways. It is designed to operate over a wide-area network, assuming no underlying remote execution system. Thus, workers operate autonomously, pulling work units from the master. It does not manage the allocation of the workers themselves, but it does provide for security and for operation through network disconnections.

An XW worker manages a machine in a manner similar to a Condor startd. It only permits remote processes to run within the constraints of the machine's owner. For example, it may permit remote users to harness the machine at night or when the console is idle. When available to perform work, the worker contacts a well-known master to obtain a work unit. It is then free to operate while disconnected and may choose to pause or discard work at its discretion. When disconnected, a worker is responsible for sending periodic messages to the master to indicate that it is still alive.

The pull nature of the XW worker is well suited to the realities of the current Internet. Firewalls, address translators, and similar devices often prevent symmetric network connectivity. However, many such barriers permit a TCP connection to be initiated from the private portion to the public network, thus encouraging a design in which the master is placed on a public network and passively accepts incoming connections from workers, which are more numerous and likely to execute on private networks.

XW provides security mechanisms to protect the integrity of the application and the machine owners. For portability, the worker itself is bootstrapped in a Java virtual machine, but end-user computations may be performed in native code. To protect the owners of worker machines, only native code certified by a central authority is accepted for use. A public-key encryption system is used to ensure code authenticity and to protect communications between the master and workers.

Because the XW master manages only work and not the workers themselves, an application has no direct control over the number of workers that it harnesses. However, this task can be performed outside the context of the system. For example, an XW system may be integrated with a Condor pool or Condor-G system by starting an XW master and submitting a number of XW workers to a schedd. As Condor is able to execute the XW workers, they report to the known master, distribute the application code, and go to work.



XW is planned to be used as a distributed resource manager for the Pierre Auger Observatory. This international scientific group aims to study high-energy cosmic rays indirectly by way of their interaction with the Earth's atmosphere. To properly calibrate the detector, its behavior to a large number of incident rays must be simulated. Approximately 10 CPU-years (570) of simulation must be performed. To complement the existing standard computing clusters, XW will be used to distribute the FORTRAN simulation code to opportunistic workers.

19.4 THE LAYERS OF GRID COMPUTING

The preceding discussion shows how a complex Grid computing system can be built up in layers, with each layer assuming the responsibility to multiplex resources for a different reason. Figure 19.11 shows these responsibilities. Together, the Condor-G gridmanager and jobmanager form an *inter-Grid* system. They are responsible for bridging the gap between autonomous systems on the Grid and are concerned with matters of security and disconnected operation. They have limited facilities for managing different classes of machines and jobs. These problems are better solved by the Condor schedd and startd, which form an *intercustomer* system. Once connected via the inter-Grid system, they manage all the details of heterogeneous systems and arbitrate requests for service from different customers. However, they do not provide services for managing the order and assignment of work to be done, tasks that are better handled by an *intertask* system such as DAGMan, Condor-MW, or XW.

The most flexible layers of a Grid system are task-parallel. The unique feature of a task-parallel system is that it provides separate interfaces for the allocation of resources and the assignment of work. For example, in both Condor-MW and XW, the user may manipulate the number of workers available,

Master Process		**Worker Process**
Gridmanager	**Inter-Grid Layer**	Jobmanager
Schedd	**Inter-Customer-Layer**	Startd
MW-Master XW-Master	**Inter-Task Layer**	MW-Worker XW-Worker

19.11 FIGURE The layers of Grid computing.

by either calling a procedure (Condor-MW) or invoking new workers externally (XW.) This distinction leads to a powerful and robust system design, because the concerns of performance are separated from those of correctness. If higher throughput is needed, more workers may be added. If shorter latencies are needed, nearby or faster workers may be added. The master's work-assignment algorithm is oblivious to what resources are selected, and persists through all failures.

The same distinction between allocation and assignment is found in other layers as well. The traditional Condor intercustomer system is task-parallel at heart. The schedd allocates machines using the matchmaking algorithm and then only assigns work in the order of local priority as machines become available. Like a Condor-MW or XW master, the schedd may hold allocated machines idle as hot spares or as a cache for future requests, if it is willing to pay the price.

When allocation and assignment are coupled, the system is less flexible. For example, Condor-G couples machine allocation with job execution. It submits a job to an existing remote batch queue without advance knowledge of whether the job will execute immediately or at all. If multiple queues are available, it must send a job to wait in one queue or the other and may find it difficult to enforce local orderings on job execution. To remedy this problem, we may layer a task-parallel system on top, as is done with gliding-in. The task-parallel Condor system multiplexes the job-parallel Condor-G, allowing the user more flexibility in resource allocation.

Thus, rather than place all of these responsibilities in one process or system, each layer is responsible for the needs of one type of element: Grids, customers, or tasks. Depending on the desired functionality, users can assemble an appropriate system by overlaying one layer on top of another.

Exactly such a layered system was built in the year 2000 to attack a series of unsolved optimization problems (443). Using Condor-MW, four mathematicians from Argonne National Laboratory, the University of Iowa, and Northwestern University constructed a branch-and-bound searching algorithm. This technique divided the initial search space into smaller regions, bounding what could be the best possible solution in each. Despite a highly sophisticated solution algorithm, the time required to find a solution was still considerable: over seven years with the best desktop workstation available at the time. This solver was deployed on a layered system consisting of the master–worker framework running on a traditional Condor pool glided in over a Condor-G system. A solution to the NUG-30 facilities assignment first posed in 1968 was found in less than one week by using over 95,000 h of CPU time on 2,500 CPUs at 10 different physical sites.

19.5 RELIABLE OUTPUT

The final problem that we consider is the following. Jobs placed on remote execution sites may wish to read and write from archival storage, send messages back and forth to other running processes, or read and write intermediate results at known rendezvous points. These activities are known collectively as input and output (I/O). Regardless of destination, most I/O activities are performed via standard file-system operations such as open, read, write, and close. As with remote execution, these operations assume a fault-free connection between the application and the target storage device, and reliability is at best an illusion provided to the application. Within a Grid computing system, we must be prepared to deal with all manner of failures.

We focus here on output. Input is a well-studied problem: file systems, databases, and Web browsers have all addressed the difficulties of replicating read-only data for the sake of a migrating client (254, 528). Much of that research is relevant and already implemented for Grid computing systems. Furthermore, output has stricter requirements for consistency. Whereas an already-read input may be discarded with the knowledge that it may be re-read at a predictable cost, output data are usually (but not always) a more expensive commodity produced by a careful confluence of inputs with software and hardware. Indeed, some outputs (such as those of a physical measurement) may not be reproducible at all. Finally, output may be considered the simple dual of input. Every reader requires a writer, and the recording of data into stable storage is more complex than its retrieval.

We assume that a running application wishes to record its output in a named storage location. Whether this is an archival device, rendezvous storage, or another running application is a minor detail. The simplest way to accomplish this task is to connect the file-system operations of an application by remote procedure call to the storage device. This is known as remote output. As we will see, this method is sensitive to all of the problems of disconnection discussed previously in the context of the remote execution scenario. We present several techniques for making remote output robust to system failures. However, before we embark on that discussion, we must establish several primitives for reliably recording output data in stable storage.

19.5.1 Storage Transactions

Just as execution systems require careful protocols such as two-phase commit to provide a reliable service, so too do storage systems require disciplines for ensuring that complete actions are secure and incomplete actions are cleaned

up. However, unlike databases, most mass-storage devices do not support the complete notion of a transaction as we explored earlier. Instead of begin, prepare, commit, and abort, we have only *fsync*, which forces all previous changes to storage. Nevertheless, we can still perform reliable storage operations if we accept limitations on either the sort of operations performed or on the nature of the transaction itself.

If we limit programs to performing only the storage of complete new files, we can provide reliability through what we call a *delivery transaction*. This discipline provides the same guarantees as the ordinary two-phase commit protocol. Suppose that a program wishes to deliver a new file f1. To begin a delivery transaction, a new file f1.t must be created. The program may then write data into f1.t in any desired order. To prepare the transaction, the write permissions on f1 are checked, the data are fixed to permanent storage by invoking fsync on f1.t, and the file is renamed to f1.p. Finally, to commit the transaction, the file is renamed into f1.

This discipline satisfies the two-phase commit requirements as follows. If the transaction is to be aborted at any time, f1.t may simply be deleted without harm. If either the client or server should crash before the transaction is prepared, the recovery procedure should delete all unprepared transactions by deleting any file ending in .t. Files ending in .p have been fixed to disk and may safely be retained. Finally, if either side should crash while a rename is in progress, the POSIX (32) semantics of file renaming ensure that the entire file will exist under exactly one of the names f1 or f1.p.

Delivery transactions are certainly a dominant mode of interaction in Grid computing, but they are not the only mode. Some user's jobs require more sophisticated interactions with storage, sometimes creating and deleting multiple files. A limited form of reliability may still be provided for these jobs without resorting to a full-blown database system. If we limit a job to the use of certain operations and apply the fsync operation as a form of commit, we can provide what we call a *half-transaction*. A half-transaction guarantees that a series of operations, once committed, are safely on stable storage. A half-transaction has no facility for aborting incomplete work. It must be carried forward.

When using half-transactions, we must limit a job to idempotent operations. An operation is idempotent if it may be run multiple times, perhaps not even to completion, but if the final execution is successful, then the end effect on storage is the same as if exactly one successful execution occurred. Most file-system operations, if correctly recorded, are idempotent.

For example, the creation of a file by name is idempotent. It may be repeated many times and the file still exists exactly once. Writing data to a file is also idempotent if the write offset is explicitly given. A few key operations are not

idempotent. For example, appending to an existing file is not idempotent if the current file size is the implicit write point. Renaming a file from one name to another is also not idempotent. These operations must be avoided when using half-transactions.

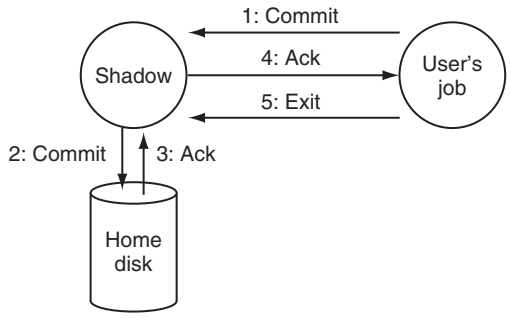
Half-transactions are used as follows. A stream of idempotent output operations may be written to storage. To complete the half-transaction, a commit must be issued. A standard file system provides commit by allowing a client to invoke fsync on all files that have been modified. If the client receives an acknowledgement for the commit, then the half-transaction is complete. If the connection between the client and server is lost, then the entire half-transaction must start again. However, it does not matter if any or all of the write operations have succeeded or failed, because the idempotent property guarantees that the end result will be the same. When the likelihood of failure is high, a single half-transaction may be strengthened by adding periodic commit operations before the final commit. If a failure occurs, the half-transaction need only be resumed from the operation following the most recent commit.

19.5.2 Reliable Delivery Techniques

Whether we are performing delivery transactions for whole files or half-transactions for arbitrary updates, Grid computing requires reliable services for delivering the inputs and outputs. However, it is difficult to divorce the completion of output from job execution. The manipulation of storage is an important side effect of execution, and we would negligent indeed if a job completed reliably without performing its output (or vice versa). Thus we must pay special attention to the integration of job completion with output completion. Where the word commit appears, it may be used to indicate the completion of either a storage or half-transaction, whichever is in use.

The most natural form of output is *direct output* (Figure 19.12). In this model, a job stays in close contact with its target storage. When it requires input or output, it issues a remote procedure call to the target storage to access some limited portion of the data that it needs immediately, perhaps with some ability to cache and buffer expected data. This approach is used in the Condor remote system call facility (602), in which every system call executed by a job is sent home for execution at the shadow. Similar techniques are found in distributed file systems such as NFS (572) and AFS (370).

No matter what type of storage transaction the job employs, it is responsible for seeing that any modifications run to completion. Thus, the job itself is responsible for issuing a final commit to the target storage before indicating a



19.12 FIGURE

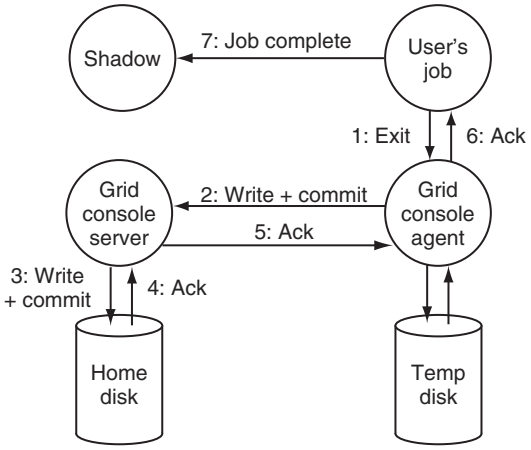
Direct output via remote system calls. When the job is ready to exit, it issues commit to the shadow. The shadow forces all output to the disk, and sends an acknowledgement back to the job. Now satisfied, the job informs the shadow that it is complete.

successful exit to the process that supervises it. In traditional Condor, the roles of storage device and supervisor are both played by the shadow process, but the same principle applies to other systems. If the commit does not succeed, the job is responsible for aborting itself rather than indicating a successful completion.

The direct output technique is easily constructed and well understood. However, it has one major drawback. It holds the job hostage to the whims of the larger system. If the storage device crashes, the network fails, or a temporary load diminishes bandwidth, the job will pause as it waits for the remote procedure call to finish. This is the case regardless of whether the output is performed over a stream connection, as in Condor, or over datagrams, as in NFS. On the other hand, the output might complete, but the job might fail or be evicted before it can record a successful completion.

One technique for insulating the job from these problems is to perform all I/O in large pieces before and after the job executes. This strategy is used, for example, in the Globus Toolkit GASS system (117). It does provide limited insulation, but also creates new problems. A job's I/O must be known before it runs, which is not always possible. Also, the user cannot see progressive output, as may be required to determine whether a computation is running correctly. Some jobs may permit limited interactive steering when run in a batch mode, so interactive access may allow more efficient use of resources.

What many jobs and users require is an "interactive when possible" I/O capability that makes a job's inputs and outputs immediately available *when network conditions permit*. Otherwise, output is buffered and performed in the

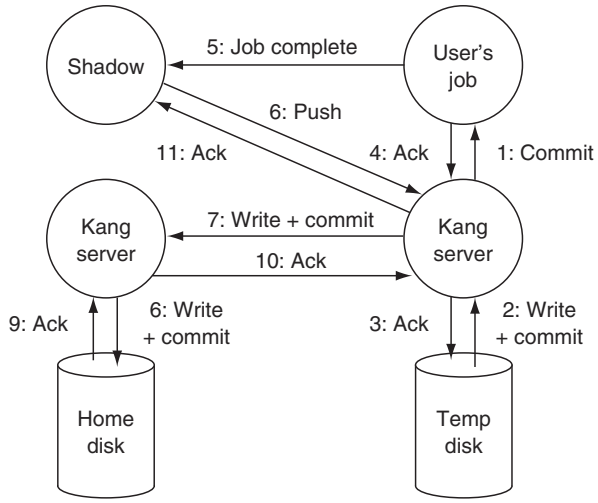


19.13 FIGURE

The Grid Console permits more flexible I/O coupling. As the job runs, it writes output data to the agent, which stores it on the temporary disk or forwards it to the server. When the job wishes to exit, it informs the agent, which must then force all output to the server. If successful, the job is informed and may indicate completion to the shadow.

background so as not to unnecessarily delay execution. The *Grid Console* provides this capability. It is composed of two parts, an *agent* and a *server* (Figure 19.13). The agent attaches itself to the job's standard input and output streams using the Bypass (638, 639) interpositioning tool. As the job runs, it traps I/O operations and, if the network is available, passes them to the shadow via remote procedure calls. If the network is not available, output data are simply written to local temporary storage. When the network becomes available again, the agent writes the buffered outputs to the shadow and resumes normal operations. Input data are read on demand from the server. If the network is not available when an input is needed, the job is blocked. When the job wishes to exit, the agent ensures that all buffered data are written and committed, and then permits the job to exit.

The Grid Console is an improvement over direct output, as it permits the job to operate even when disconnected from the shadow. This technique is similar to the form of disconnected operation found in file systems such as Coda (413), which permit a client to optimistically operate when disconnected from the home file system. However, when the job exits, it may still be blocked if the network is down: a necessary consequence of the small but vital coupling between the job's exit code and its output side effects.



19.14 FIGURE

Kangaroo permits even more flexible coupling. The job writes its output data to the nearest Kangaroo server as it runs. When it wishes to exit, it issues a **commit** to the nearest server. If successful, it indicates completion to the shadow. The shadow must then issue a **push** in order to ensure that all output have successfully been delivered.

We may stretch this coupling even further by using the *Kangaroo* (636) I/O system, shown in Figure 19.14. A Kangaroo system is a peer-to-peer network of identical servers, each providing temporary storage space for visiting jobs. A job may queue output requests as messages at any Kangaroo server, addressing them with the name of the user's home storage. Like Internet mail, these requests proceed to their destination asynchronously, perhaps passing through several servers along the way. Unlike Internet mail, Kangaroo provides two distinct consistency operations. When a job writes data to a nearby server, it uses a commit message to make sure that the data are safe on local storage. Thus, the job may exit with confidence that the data will arrive eventually, although it does not make any guarantees about exactly when it will arrive. The job informs the shadow that it has exited and evacuates the machine, making it available for another job to use. The shadow, knowing the output may still be spooled at the remote Kangaroo server, must use the second consistency operation, push, to ensure that all dirty data are forced back to the home storage site before informing the user that the job has completed.

These three variations on reliable output share many properties with remote execution. The simplest systems involve tight coupling between the submission

and execution sites. A more relaxed coupling has advantages, but it demands more complex interactions between client and server.

19.6 FUTURE DIRECTIONS

We have introduced techniques that a client of distributed services can use to deal with disconnections and failures and to ensure that remote services reach an acceptable state. (Just as database system imposes order on unstructured disk drives, so too must Grid clients impose order on unstructured workers.) We conclude by reflecting on the properties of an ideal remote Grid service, one that would be a reliable and effective partner for a Grid client. Although remote services need not be perfect (good clients are prepared to deal with failures), better services can enable more effective and efficient Grid clients.

An ideal remote execution service would have:

1. Powerful ways to asynchronously interact with third parties, such as databases or file systems, on behalf of a job.
2. Precise tools for measuring and limiting what resources a job consumes.
3. Reliable methods for cleaning up an unmanaged machine before and after each job executes.
4. Clear responsibility for dealing with failures and unexpected events without continuous oversight from the submitter.

We call this imagined reliable partner the *Grid Shell*. A Grid Shell surrounds a job, insulating it from a hostile environment. Some of this insulation occurs by preparing the execution site appropriately, perhaps by transferring files or installing other software before the job runs. Some might also occur by emulating the necessary services at run-time. Whatever the tools available to the Grid Shell for dealing with the complexity of the outside world, it must rely entirely upon an external policy manager (such as a shadow or a gridmanager) to control how and where it uses remote resources.

A Grid Shell does not exist now, although there already exist many pieces of software that provide some of the needed capabilities. Like a traditional command-line shell, it does not perform any of the user's work directly, but is an overseer or caretaker responsible for watching a user's job throughout its lifetime. It must locate an executable from a number of choices, given a particular user's preferences and settings. It must set up physical input and output objects

and bind them to logical names such as standard input and output streams. It must create the job process and then monitor its completion, detecting whether the job has completed normally or was simply unable to execute in the current environment.

Although a Grid Shell would generally not be an interactive tool, it must follow user commands and policies. As a user will not typically be physically involved in every decision made, the Grid Shell acts as the agent or representative of the user and controls resource consumption decisions made for the user. In the Condor, this role is filled by the shadow. Although other components perform all of the complicated roles necessary to carry out distributed computing, the shadow alone is responsible for simply directing what is to be used. The Grid Shell must respect this distinction.

For example, the Globus jobmanager satisfies the first requirement for a Grid Shell. It transfers all necessary standard input and output streams before and after a job executes. However, it does not decide where these streams come from on its own. It is informed of this information by the gridmanager, which fulfils the role of the shadow in a Condor-G system. The jobmanager is also an excellent tool for disconnected operation, as it is capable of moving these streams asynchronously and resuming after a failure.

The second requirement is satisfied by the traditional Condor starter. As a job runs, that starter continuously monitors all of its resource use, including CPU time, memory size, and file-system activity. If the job exceeds any limits set by the user, then the starter can cleanly terminate the job. This capability can be used to prevent runaway consumption due to software or hardware failures, or it may simply limit the user's consumption to what is mandated by the community as his or her fair share.

A traditional Condor startd meets the third requirement. When a disconnection or other failure prevents a job from running properly, a Grid Shell must be careful to precisely clean up the execution site in a manner expected by the submitter. This frees the submitter to attempt the job elsewhere without fear of creating conflicts between multiple running jobs and without unnecessary consumption of resources. Of course, a Grid Shell need not be aggressive about cleanup. If the submitter and the shell agree, an appropriate time interval may pass or other conditions may be met before cleanup occurs. As long as the two parties agree on the conditions, a reliable system will result.

The final requirement states that the shell must have some independent facility for retrying failures without oversight from the submitter. We have explored this facility with an experimental piece of software known as the *fault-tolerant shell*. This is a simple scripting language that ties together existing programs using

an exception-like syntax for allowing the failure and retry of programs. For example, this script attempts a file copy followed by a simulation for up to five attempts at one hour each. If either should fail, the pair is tried again until a time limit of three hours is reached:

```
try for 3 hours
try 5 times
  copy-data ftp://ftp.cs.wisc.edu/data data
end
try for 1 hour
  run-simulation data
  end
end
```

We imagine the Grid Shell as an ideal partner for remote execution in a Grid computing environment. Many of the properties of the Grid Shell have been explored in existing software, but work remains to tie these disparate systems together into a coherent whole. If carefully engineered with public interfaces, such a tool would be useful to all manner of remote execution systems, regardless of the vendor of the client or server.

19.7 SUMMARY

Grid computing is a partnership between clients and servers. Grid clients have more responsibilities than traditional clients, , and must be equipped with powerful mechanisms for dealing with and recovering from failures, whether they occur in the context of remote execution, work management, or data output. When clients are powerful, servers must accommodate them by using careful protocols. We have described some of the algorithms that may be used to help clients and servers come to agreement in a failure-prone system. Finally, we have sketched the properties of an ideal service for remote execution called the Grid Shell.

Many challenges remain in the design and implementation of Grid computing systems. Although today's Grids are accessible to technologists and other determined users willing to suffer through experimental and incomplete systems, many obstacles must be overcome before large-scale systems are usable without special knowledge. Grids intended for ordinary competent users must be designed with as much attention paid to the consequences of failures as the potential benefits of success.

ACKNOWLEDGMENTS

This work was supported in part by Department of Energy awards DE-FG02-01ER25443, DE-FC02-01ER25464, DE-FC02-01ER25450, and DE-FC02-01ER25458; European Commission award 18/GL/04/2002; IBM Corporation awards MHVU5622 and POS996BK874B; and National Science Foundation awards 795ET-21076A, 795PACS1077A, 795NAS1115A, 795PACS1123A, and 02-229 through the University of Illinois, NSF awards UF00111 and UF01075 through the University of Florida, and NSF award 8202-53659 through Johns Hopkins University. Thain is supported by a Lawrence Landweber NCR fellowship and the Wisconsin Alumni Research Foundation.

FURTHER READING

- Chapter I:14 and a series of technical articles describe early Condor development (444, 446, 489), checkpointing (602), and remote system calls (445).
- Other articles describe the matchmaking model and ClassAd language (538, 539, 541), as well as extensions to the ClassAd language (447, 637).
- Transactions and reliability in databases and other distributed systems are described in standard texts (327, 537); technical articles describe earlier developments (324, 326).