Transparent access to grid resources for user software

Sander Klous[a], Jaime Frey[b], Se-Chang Son[b], Douglas Thain[b],
Alain Roy[b], Miron Livny[b], Jo van den Brand[a]

[a] NIKHEF, P.O. Box 41882, 1009 DB Amsterdam, The Netherlands
[b]Computer Science Department, University of Wisconsin, United States

Abstract

Grid computing promises access to large amounts of computing power, but so far adoption of grid computing has been limited to highly specialized experts for three reasons. First, users are used to batch systems, and interfaces to grid software are often complex and different than those in batch systems. Second, users are used to having transparent file access which grid software does not conveniently provide. Third, efforts to achieve wide-spread coordination of computers while solving the first two problems is hampered when clusters are on private networks. Here, we bring together a variety of software that allows users to almost transparently use grid resources as if they were local resources while providing transparent access to files, even when private networks intervene. As a motivating example, the BaBar Monte Carlo production system is deployed on a truly distributed environment, the European DataGrid, without any modification to the application itself.

1 Introduction

At 'GRID 2002' Gabrielle Allen *et al*. made the observation that there is an eminent shortage of 'real grid users' [1]. They claim two main obstacles prevent applications from benefitting from grid computing. First, the absence of a vision how to implement this new technology and second a toolkit of higher level grid services, tailored for the application needs, is not available. They introduced the GridLab project aimed at delivering an API through which applications access and use distributed resources.

We fully acknowledge these two problems and the need for projects that enable straight forward development of new applications suitable for distributed systems. However, the idea that almost every existing application ought to be restructured to be deployed on a distributed system comes at an enormous cost in development and debugging labor. Regardless, a large number of users hopes to benefit from large-scale distributed computing. If a way is found to deploy applications on a distributed system without making modifications, then both hardware and human resources can be used more efficiently.

How are we to make distributed computing accessible to ordinary computing applications, that so far struggle to benefit from the promises of distributed computing? Typical developers write their applications on standalone machines, making liberal use of complex and powerful libraries and programming environments. By re-using existing tools, developers are able to concentrate on their craft rather than reinventing computing from the ground up. Software is created, debugged, and validated on ordinary workstations long before any thought turns to distributed computing. These issues are the challenge of distributed computing in the real world and lead to the shortage of grid users: for social and technical reasons, little attention is paid to the distributed nature of resources during the development of most user applications.

The solution we propose is two fold. First, the users interact with a well-known batch interface that hides the complexities of a system that spans various administrative domains. Although this batch system provides familiar functionality for the end users, its implementation is unconventional: it harnesses multi-domain resources as if they all belong to one giant local computing pool. Second, the application runs in a virtual environment that provides it with the illusion of a local system, independent of the actual location of the resources it uses.

These objectives are achieved by integrating the functionality of five components. The first component is **'Condor'**, a batch system providing a standard user interface to manage distributed resources. The second component is **'Condor-G'**, a tool that uses inter-domain resource management protocols to provide resource discovery and resource access across administrative domains. The third component is called '**gliding in**', a technique that allows the remote resources to become an integral part of the original batch pool. '**Generic connection brokering**' is the fourth component, making it possible to use the other components across firewalls and private networks. The last component is **'Parrot'**; it implements an interpositioning technique based on the debugger trap to provide the application with transparent file I/O over a wide area network without any modification to the user application.

As a motivating example, the deployment of an applica-

tion into a real distributed environment is described, without any modification to the application itself. This application, the BaBar Monte Carlo production system, is representative of the applications described earlier; it consists of multiple processes and complex libraries, poorly suited to most distributed computing and file systems. In particular, the problem of working through an aggressive firewall that discards active TCP connections is addressed.

2 Working on a computational grid

There exist countless systems for harnessing remote processors and accessing remote data, but many place stringent requirements on the applications that they accept. A batch system might require that all programs be a single executable performing no interprocess communication. A distributed file system may provide unusual consistency semantics that are at odds with a user's expectations. Many experimental systems expect users to re-write their software to take advantage of new features, while many production systems expect users to have administrator privileges on all machines on a network.

A computational grid is inherently an unfriendly environment with its own challenges. Significant problems arise when resources need to be discovered, acquired and managed across the boundaries of administrative domains. Furthermore, installing most software on a new cluster is a labor-intensive process that defies automation: executables, scripts and libraries must be unpacked and installed; environment variables and other settings must be configured; database structures must be initialized; dependent software must be discovered and installed. Some software expects a uniform user database across multiple machines; this is an impossibility on a computational grid. The nature of a distributed environment ensures that network outages and performance variations are common events.

Users and applications should be shielded from these complications. The satisfaction of a user with grid computing depends heavily on the amount of effort required to deploy an application on a distributed system. This effort is drastically reduced when users can submit their jobs through a well-known batch interface, one that does not require extra knowledge to run an application on resources in different administrative domains. The assembly of such a system is presented in section 2.1. The way this 'overlay batch system' deals with network complexity is entirely different from that of a standard batch system. In contrast to normal batch nodes, grid resources can often not be accessed directly by the batch system. These connection issues are handled by a generic layer in between the batch system and its system calls as will be discussed in section 2.2.

Another equally important factor in the user experience is the success rate of the remotely executed applications. In section 2.3 the construction of a virtual environment is discussed that allows almost any application that runs correctly on a local machine to complete successfully on grid resources as well.

2.1 Resource management

The proposed user interface to the grid resources is organized around **Condor**, a batch system to manage the workload [2]. It provides a job queuing mechanism, a scheduling policy, a priority scheme, resource monitoring and resource management. The users submit their jobs to the batch system, which places them into a queue. The batch system chooses when and where the jobs are run based upon its policy and monitors their progress. Ultimately the user is informed upon completion.

Consider a grid environment in which an individual user may, in principle, have access to computational resources at many sites. The user defines a job on a user interface machine by specifying the resource requirements, the input and output files and the program to be executed. An agent runs on a machine with access to the batch system and provides a reliable single access point to all the resources the user is authorized to use. **Condor-G** [3] contains an implementation of such an agent. The functionality of Condor-G is shown in Fig. 1. It uses the protocols defined by the Globus Toolkit [4], a de facto standard for grid computing, to communicate with the grid sites.

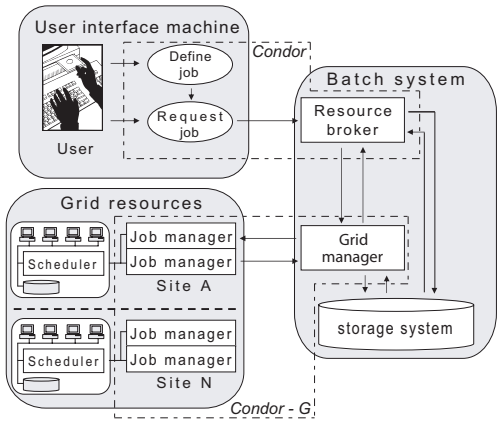

Figure 1: Condor-G functionality. The batch scheduler creates a grid manager to handle jobs on remote resources.
Note that these figures use the following convention: tasks are in circles, processes in boxes and groups in boxes with rounded corners.

The scheduler of the batch system responds to the request to run jobs on grid resources by creating a new grid manager process to submit and manage those jobs. One grid manager handles all jobs for a single user and terminates once they are complete. All authentication requests are handled by Condor-G via the Grid Security Infrastructure (GSI, [5]), which allows for single sign-on based on a Public Key Infrastructure (PKI). The grid manager submits the jobs to

a grid site using the Grid Resource Allocation and Management (GRAM, [6]) protocol. On the grid sites, each received job results in the creation of a job manager process. This process handles the job life cycle at the grid site. It connects back to the grid manager, with GSI mechanisms for authentication and retrieves the job's executable and input files using Global Access to Secondary Storage (GASS, [7]). The standard output and error of the jobs are streamed real-time via the same route. The process submits the job to a scheduler running on the grid site and keeps the grid manager up to date about the job status. The grid manager in turn informs the batch system on the user site to make sure the user gets an accurate overview of the jobs on the remote resources.

The techniques described above allow for the construction, submission and monitoring of jobs on remote resources. Condor-G provides fault tolerance that handles the complications of a truly distributed environment. Still, it does not provide a complete integration of the grid resources with the batch system on the user site. Jobs submitted to various grid sites end up in different queues, often with different types of scheduling policies and priority schemes. The user does not know and does not want to know the details of the resource allocation mechanisms for all these sites to get information about the expected time the job will run. An overview of allocated resources that can immediately be used to run the jobs listed in a user queue is much more convenient. The jobs in this queue are awaiting either the allocation of new resources or one of the already allocated resources to become available. This functionality is implemented with a technique called 'gliding-in' [3], which is shown in Fig. 2.

In the **gliding-in** scheme, Condor-G no longer submits user jobs to the remote resources. Instead, the submitted jobs contain the batch system processes themselves. The required software for these jobs is downloaded from a central storage with a GSI enabled file transfer protocol (GSIFTP, [8]). When the processes are running on the remote grid resources, it will appear as if the batch system on the user site has grown. Users can now submit their jobs to these resources directly via their batch system, without the use of grid mechanisms. In this sense, gliding-in creates an overlay batch system: the acquired grid resources provide all of the features of normal batch nodes.

As a positive side effect, this method disentangles the stages of allocation and execution. The batch system can guarantee optimal queuing times to its users by submitting batch processes to all remote resources, annulling the ones it does not use. This prevents a job from waiting at one remote resource while another resource capable of serving the job is available[1]. Batch processes on remote sites shut down gracefully when they do not receive any jobs to execute after a (configurable) amount of time, thus guarding against runaway processes. Resource allocation can further be optimized with the development of more intelligent resource planning and scheduling agents, which are able to negotiate with grid scheduling tools.

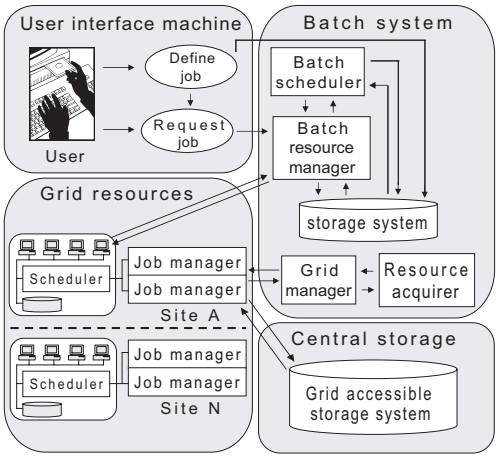

Figure 2: Gliding-in functionality. The remote resources become an integral part of the pool, directly managed by the batch system on the user site.

2.2 Dealing with network complexity

The effort to achieve this wide-spread coordination of computers is hampered when clusters are on private networks. Batch systems in general assume the availability of bidirectional communication to the worker nodes. Although worker nodes on grid sites often have outbound IP connectivity, problems arise when the batch system on the user site tries to initiate contact with the worker nodes on private networks or behind firewalls. Private networks are however common practice, since they provide easy network management and address planning as well as a solution to the IPv4 address shortage problem. Firewalls play important roles in protecting networks and are ready to play an even more important role as the security headquarters of integrated security systems.

For the batch system to work seamlessly across private networks and over firewalls the symmetry in the peer-to-peer connections needs to be recovered. The solution should be scalable and not require any special privileges on the side of the grid resources. **Generic Connection Brokering** (GCB) [9] possesses these properties. An example of a system using GCB is shown in Fig. 3.

The system shown consists of one node in the public and two in the private network that try to run applications requiring bidirectional communication between them. The applications are GCB-enabled on both the public and private nodes, *i.e.* they are able to use GCB functionality to

[1]This can also be accomplished by directly submitting the user job to multiple sites. However, the separation of the allocation and execution stages allow for a more opportunistic scheduling approach. It is not necessary to have the user job available when a request for resource allocation is submitted.

solve communication issues. Outbound IP connectivity of the nodes on the private networks is crucial for this solution[2]. The nodes in the private network send a register request to the GCB server (solid lines). The server creates proxy sockets of the same type as the client sockets, binds them, makes them passive and returns the addresses to the private nodes. From now on, the private nodes use these addresses as their network identity (*i.e.* whenever they need to inform other processes of their address, they send the proxy address instead of their real address). When the public node wants to connect to the private node, it asks the GCB server to broker the connection. The server decides, based on the network situation of both nodes, who should actively connect and arranges accordingly (dotted lines). If either cannot connect to the other, because *e.g.* both are on private networks, it lets both parties connect to the server and relays the packets between them (dash dotted line).

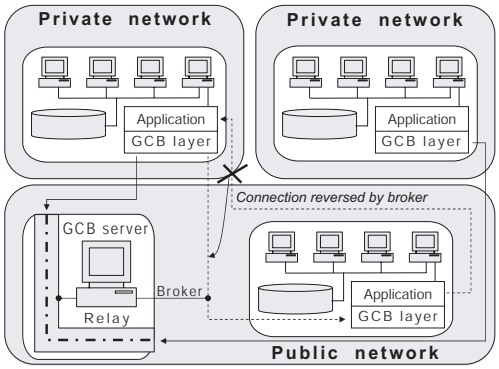

Figure 3: GCB functionality. Connections involving nodes on private networks are rearranged when necessary.

Gliding-in and GCB form a powerful combination in the area of grid computing. Together they are capable of extending the functionality of a batch system on the user site into private networks of other domains without the need of administrative privileges. GCB is implemented as a layer between the batch system software and its system calls. Since the GCB layer provides the same interfaces and semantics of socket calls, the batch system software can normally be linked with GCB without modifications[3].

The GCB layer checks every incoming message and decides to either pass it to the batch system process, or handle it appropriately if it is a GCB command. GCB can use both the TCP and UDP protocols for communication. In our experience the end to end reliability of TCP makes this protocol preferable over the UDP protocol in hostile environments. Although the UDP protocol is in principle superior in performance, it silently fails to pass the stringent requirements of the active components in some private networks.

The GCB server is implemented as a process that can run with standard user privileges. Since the server does not assume the responsibility of initiating connections to the clients, it can be placed anywhere on the public internet. It maintains accurate information about the status of the clients using the heartbeat messages they send periodically. The server is the weakest point in the system described. It maintains the connections to all machines in the private networks, and represents both a single point of failure and a security concern.

GCB may be deployed to any grid site that allows only outbound connections from the worker nodes without intervention of network administrators. This stretches site security policies, maybe even to an extent that allows for abuse. Security however, should be provided in an orthogonal way to connectivity, and developments are on their way to provide GCB with a strong security mechanism.

Resources managed by the GCB server can span multiple private networks. This means that there is a risk of private IP address collisions, since this version of GCB uses the standard TCP/IP addressing scheme. An extended addressing scheme is implemented in a new version of GCB (extended or eGCB) to uniquely identify all machines.

The batch system on the user site was successfully extended across various administrative domains, private networks and firewalls with the techniques discussed in this section. Although this allows for easy resource management, it does not provide transparent access to files. For this an interposition agent can be used, which allows modification of the file I/O of any application, without modifying the application itself.

2.3 Remote execution with interposition agents

An interposition agent is a piece of software that inserts itself between two existing layers of software in order to modify their discourse. By inserting an interposition agent rather than modifying an existing piece of software, we may measure, debug, and enhance an application without requiring intimate knowledge of its innards. An interposition agent has many uses in a distributed system:

Seamless integration. The most common use of an interposition agent is to connect an application to a new resource, such as a storage device, without requiring any special changes or coding in the application. For example, an interposition agent can allow an application to seamlessly connect to a remote storage server. The application merely perceives it to be an ordinary file system.

Improved reliability. In general, remote data services are far less reliable than local file systems. Remote services

[2]This statement is not entirely correct. The nodes on the private networks need to be able to contact the GCB server and the GCB server needs to be accessible via a public internet address. If the GCB server manages only one private network, it could be installed on the head node and the private nodes would not need outbound IP connectivity. This construction would however require the ability to start a long-running process on the head node.

[3]Note that GCB is only linked to the batch system software, which does the resource management. The user application will run exactly as it is.

are prone to failed networks, power outages, expired credentials, and many other problems. An interposition agent can attach an application to a service with improved reliability. For example, it can emulate a reliable TCP connection across network outages and address changes or add reliability at the file system layer by detecting and repairing failed I/O connections. When combined with generic connection brokering as discussed in the previous section, it can even provide bidirectional communication for peer-to-peer applications across private networks and firewalls.

Private name-spaces. Batch applications are frequently hardwired to use certain file names for configuration files, data libraries, and even ordinary inputs and outputs. An interposition agent can be used to create a private name-space for each instance of an application, thus allowing many to run simultaneously while keeping their I/O activities separate. For example, several instances of an application hardwired to write to "output.txt" may be redirected to write to "output.n.txt", where n is the instance number.

Remote dynamic linking. Although dynamic linking offers many technical advantages for programs that share code or data, it presents a number of practical problems. It is all too easy to migrate an application only to discover that needed libraries are missing, or worse yet, that the available libraries are the wrong version. An interposition agent can solve these problems by allowing an application to link against libraries stored at a single, well-known server.

Profiling and debugging. The vast majority of applications are designed and tested on standalone machines. A number of surprises occur when such applications are moved into a distributed system. Both the absolute and relative cost of I/O operations change, and techniques that were once acceptable (such as linear search) may become disastrously inefficient. By attaching an interposition agent to an application, a user may easily generate a trace or summary of I/O behavior and observe precisely what the application does.

Parrot is an interposition agent that provides the features discussed above for standard Unix applications [10]. It observes and potentially modifies the interaction between an unmodified process and the operating system kernel using the standard Linux ptrace debugging interface, which traps all system calls of the process. When used in an unfriendly distributed system, Parrot provides the illusion of a user's home environment, including files, user identities and more. It can customize an application's environment to create a synthetic name-space formed from multiple remote services. In addition, it is able to hide network outages, server crashes and other failures that are endemic to distributed systems.

Although the notion of interposition agents is not new, they have seen relatively little use in production systems. This is due to a variety of technical and semantic difficulties that arise in connecting real systems together. For example, many different I/O protocols may be attached to an application, but few provide the full range of POSIX semantics expected by many applications. For this reason a dedicated protocol was created, Chirp[4], which provides the precise semantics that applications expect. Each Chirp operation is a remote procedure call from a client to a server. A Chirp operation is initiated by a client, which sends a formatted request. The server acts upon the request and sends a response. It is assumed that Chirp is carried over a stream protocol such as TCP. Authentication and authorization can be done through a variety of methods. Most interesting for the current setup is the GSI authentication as mentioned before, which allows for integration of the Chirp server with the batch system using the existing infrastructure.

Parrot is an extension to an existing operating system; it augments file-handling capabilities without affecting a process' ability to interact with other processes on the same machine or over a network. Parrot is considerably simpler than other tools like virtual operating systems such as User Mode Linux and virtual machines such as VMWare, both of which require the user to build and maintain virtual networks, large file system images, and all the elements of an isolated operating system in miniature. Parrot consists of a single executable measuring only 8.4 MB with all options enabled, and as small as 1 MB in minimal configuration.

Figure 4 shows the control flow necessary to trap a system call through the ptrace interface. Parrot registers its interest in an application process with the operating system kernel. At each attempt by the application to invoke a system call, the host kernel notifies Parrot. Parrot may then modify the application's address space or registers, including the system call and its arguments. Once satisfied, Parrot instructs the host kernel to resume the system call. At completion, Parrot is given another opportunity to make changes before passing control back to the kernel and the application.

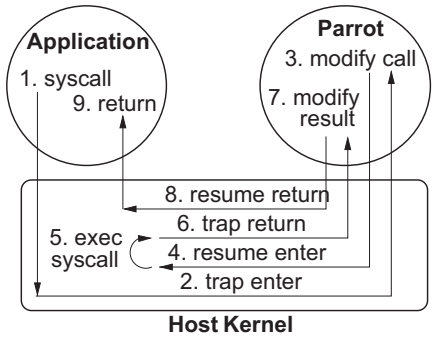

Figure 4: Interpositioning via the debugger interface.

3 An example application

The Monte Carlo production system of the BaBar high-energy physics experiment [11] in progress at the Stanford

[4] http://www.cse.nd.edu/~ccl/software/manuals/chirp.html

Linear Accelerator Center is called SP5[5]. Although the exact details of this application do not matter, it is interesting to know how SP5 operates at an abstract level. First, it loads the data that describe the configuration of the detector and the physics of particle generation. Once loaded, it enters a compute-intensive phase where it generates an arbitrary number of events that can each be summarized in 10-100 kilobytes. This means, in theory, SP5 has the right structure for distributed computing. The initial data can simply be distributed to a number of processors, production can be performed in parallel, and the produced events can be returned to a central site. Once initialized, any processor can produce an arbitrary number of events, so the number of processors can be chosen to balance startup time against desired throughput.

In practice, SP5 has a number of complexities that make it difficult to deploy in a distributed system. A standard file system contains the SP5 executable and scripts, several dynamic libraries, the input configuration, and the output events. The program is wrapped by a script that establishes environmental settings and verifies the integrity of the files before invoking the program. It also makes use of several dynamically-loaded libraries, particularly the Objectivity database, which manages the configuration and event data structures.

Objectivity is a decentralized, cooperative database built on top of a standard file system. Consistency management, access control, and crash recovery are performed cooperatively by clients rather than enforced by a server. A minimal central server assists only with a locking protocol. To read the configuration data or write events, the client library requests a lock from the lock server, manipulates the file system directly, and then releases the lock.

This structure is quite reasonable when viewed alone, but is difficult to adapt to an existing distributed system. For example, the file system activity of the Objectivity client library cannot be carried over a standard distributed file system. The delayed-writeback semantics of NFS clients are too weak for database structures, while the strict open-close semantics of AFS would result in data loss on the append-only transaction log. Objectivity does have the capability to speak NFS directly to a server, bypassing the buffer cache, but deploying this requires superuser privileges at both the client and the server; an unlikely capability in a grid computing environment. Instead, we use Parrot to make SP5 believe it is accessing Objectivity locally, while redirecting the access across the network.

3.1 Deploying SP5 on a computational grid

Figure 5 shows how the pieces of the distributed environment fit together. The configuration data and output events are stored in an Objectivity-managed file system on a well-known central server. A central lock server process assists with mutual exclusion. A number of worker nodes are used to execute instances of SP5. Access to a number of worker nodes at various institutions is obtained by way of Condor-G and the Globus toolkit as discussed in section 2.1.

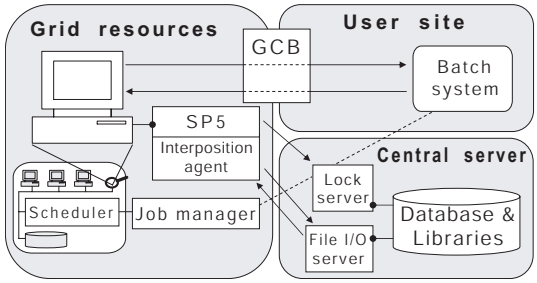

Figure 5: Deploying SP5 and Parrot on a distributed system. Communication between the batch system and the grid resources is arranged with GCB. File I/O of the user application is arranged with the interposition agent.

No special software is installed on any of the worker nodes and no superuser access is available in these environments, so Parrot needs to carry all of the SP5 file system operations back to a Chirp server run with standard user privileges deployed at the central server. Parrot makes the remote file system appear local to SP5. This is comparable to an NFS client that mounts its root file system from a remote device: all executables, dynamic libraries, and other program components are loaded from the central server via the Chirp protocol. Parrot makes local copies of executables; this is a technical necessity, because Unix can only execute a program identified by a local file name. All data files are accessed remotely without caching, to avoid consistency problems in the database.

In addition to the file system, a number of other small settings were necessary to fully emulate the home environment. For example, the Objectivity libraries examine the POSIX user identifier and host name in order to implement access control on the database. Because worker machines may not necessarily share a user database with the central server, Parrot is instructed to trap these system calls and change the results to match what would be seen at the central server.

Aggressive firewalls posed a serious problem to the deployment of this system. It is quite common for a computing cluster to be connected to the public Internet by way of a firewall and network address translator (NAT). In the clusters targeted by this application, the NAT permits cluster nodes to initiate outgoing TCP connections to the public Internet, but prohibits incoming connections. To translate external addresses into internal addresses, the NAT must keep state about every TCP connections that it carries.

The problem arises when a NAT must discard TCP con-

[5] http://www.slac.stanford.edu/BFROOT/www/Computing/Offline/Production/userguide/userguide.html.

nections that it perceives to be idle. Each connection consumes some state in the firewall, so it cannot keep them forever. The most aggressive NAT encountered discards TCP connections that have been idle for only one minute. When this happens, there is a double penalty: not only is the connection lost, but the NAT does not even return an RST packet indicating that the connection was lost. The result is that both sides think the connection is present but lossy, and retry up to their maximum timeouts, which can range from minutes to hours.

This problem was deadly to SP5. Once it initialized, the lock server connection was held open and idle, while the Chirp connection was only used for the output of each event, at intervals of slightly more than a minute. While SP5 was processing the first event, the NAT would discard the TCP connections. A short time later, the entire system would hang while attempting to write out the first event.

Although it is simple to discount this firewall as an aberrant device, reconfiguring its timeout cannot be considered a reasonable solution. For the same reasons software cannot be installed at the worker node and superuser privileges are not available, reconfiguring the network interior at will is not an option. (In fact, it was later discovered that these are the factory settings for the NAT in question. The idea of negotiating with a network administrator every time this model of NAT is encountered is not very tempting.)

One solution to this problem is to change the network endpoints to generate enough traffic to keep the NAT state alive. For example, the networking stack at the central server can be modified to send TCP keepalives at the rate of several per minute. This technique was applied in order to preserve the connection between SP5 and the lock server. However, it is unsatisfying because it requires administrator privileges on at least one end and has a system wide effect, thus all sockets are affected.

A more comprehensive solution is to make the network protocol recoverable, so that the failure of the TCP connection becomes a harmless event. For example, Parrot was modified so that a failed Chirp connection was recovered by reconnecting and reopening the needed files. With this recovery method, the Chirp connection was made *fail-fast*; hence, any delay of greater than thirty seconds was assumed to be a transient network failure and would result in disconnection and recovery. This solution is more robust than simply applying keepalives – it also tolerates the crash and recovery of the Chirp server – but could only be implemented because the Chirp protocol was dedicated for this specific implementation and thus could easily be modified.

The Chirp recovery method reveals an old problem in the design of distributed file systems. Strict POSIX semantics require that an application holds references directly to files rather than names. That is, once an application opens a file by name, it keeps access to that file even if the name is deleted or renamed. Distributed file systems such as NFS and AFS solve this problem by exposing inode numbers to clients. When recovering from a disconnection, NFS and AFS clients can be assured of access to the correct files by referring to the inode numbers. Chirp cannot do this directly; the Chirp server is implemented on top of an ordinary file system and thus can only open files by name. However, the Chirp protocol can verify that the binding between names and inodes has not changed after a recovery by simply querying inode numbers with the *stat* operation. If they have not changed, then recovery is successful. Otherwise, recovery has failed, Parrot forces the application to fail immediately, and the batch system becomes responsible for restarting it from the beginning.

3.2 Performance

After addressing the problem of recovery, issues of performance are discussed. Table 1 shows the run-times of SP5, gradually increasing the logical and physical distance between it and its data on the central server. As discussed earlier, SP5 begins with an I/O-intensive startup phase, and then settles into a CPU-intensive phase of configurable length. As the distance increases, the I/O-intensive phase pays an increasing price, but the CPU-intensive phase is relatively stable.

The first line of Table 1 shows the performance of unmodified SP5, running on the same machine as where the data reside.The application is run in 'validation mode', producing additional histograms to cross check the results. Furthermore, it produces a full debugging output so that the correctness of the output can be verified. As a result the production is approximately a factor of 5 slower than the standard production on this machine. The average and standard deviation of initialization times are shown along with the average

distance	*method*	*protocol*	*CPU*	*time to initialize*	*time per event*
local	OS	files	1 GHz	$446 \pm 46\,s$	$64\,s$
local	parrot	files	1 GHz	$668 \pm 26\,s$	$65\,s$
local	parrot	chirp	1 GHz	$777 \pm 48\,s$	$66\,s$
LAN	parrot	NFS	1 GHz	$4464 \pm 172\,s$	$113\,s$
LAN	parrot	chirp	1 GHz	$4505 \pm 155\,s$	$113\,s$
wan	parrot	chirp	2.5 GHz	$6275 \pm 330\,s$	$88\,s$

Table 1: Performance of SP5 and Parrot deployed in a distributed system.

time to process an event. It initializes in 446 seconds and then processes one event every 64 seconds. Each measurement of the initialization time is the result of 10 trials. The time to process one event is an average of 2000 events. A small numbers of outliers beyond 5 σ were attributed to unrelated network traffic and discarded. Each successive row adds one component in order to measure its contribution. The second row adds Parrot, but without any remote I/O or other features; SP5 just accesses local files through Parrot. The third row adds Chirp, but without a network; SP5 accesses a Chirp server on the same machine using Parrot. As can be expected, both Parrot and Chirp slow down initialization, but have little effect on event processing.

The fourth and fifth rows show the performance of SP5 accessing its data over a local area network (latency $130 \pm 10\mu s$). In the fourth row, SP5 is using a kernel-level NFS client to access Objectivity's files, ignoring potential consistency problems due to caching. In the fifth, SP5 is using Parrot and Chirp to accomplish the same task safely without a cache. Although initialization is an order of magnitude slower than the unmodified case, the performance of Chirp is comparable to NFS. The overhead is more a function of the network than of Parrot or Chirp.

The final row shows the performance of the complete system as depicted in Figure 5. SP5 accesses its data over a wide-area network (latency $654 \pm 50\mu s$) via the firewall as discussed above. Notice that the performance numbers are not directly comparable, as the CPU is about 2.5 times as fast as the others. However, the same qualitative result as the other lines may be seen: initialization is slow, but event processing is reasonable. Note that the opportunity to distribute jobs over many resources can decrease the turn around time of the data processing by an order of magnitude or more, even when one makes suboptimal use of remote resources. Hence, an exact comparison between identical machines is not really the issue for production software in a grid environment.

Overall, the BaBar experiment must process billions of events to complete the required simulations. In the worst case of accessing data over a wide area network, the cost of computing events equals the cost of initialization at only 70 events. Given that a typical instance of SP5 processes 10,000, the cost of remote execution, while significant, can be amortized across a large run.

4 Conclusions

The batch system provides the user with a familiar interface to distributed resources. The complexities that arise from resource management in a system spanning multiple administrative domains and private networks can effectively be hidden by Condor-G, the gliding-in technique and GCB. The GCB server is the weakest point in the system described. It maintains the connections to all machines in the private networks, and represents both a single point of failure and a security concern. Developments are on their way to provide GCB with a strong security mechanism and an extended addressing scheme to avoid IP address collisions between multiple private domains. Optimal queuing times can be guaranteed to the users by submitting batch processes to all remote resources, annulling the ones which are not used. Resource allocation can further be optimized with the development of more intelligent resource planning and scheduling agents.

Interposition agents bridge the gap between applications and systems when neither are available for modification. By raising the level of abstraction on which an application executes in a batch system, a transparent and reliable environment is provided, even in an unreliable distributed system. Deploying a complex application into a distributed system is quite feasible for an ordinary user with the tools presented in this paper, as shown with the BaBar Monte Carlo production system.

Acknowledgments

We thank Concezio Bozzi and the BaBar Monte Carlo production team for their assistance with SP5 and the production site. Furthermore, we acknowledge the valuable help of David Groep and Jeff Templon during the deployment of our application on the NIKHEF EDG testbed.

References

[1] Gabrielle Allen et al., GridLab: Enabling Applications on the Grid, Proceedings of Grid Computing, Springer, ISBN 3-540-00133-6, 2002, p. 39-45.

[2] Todd Tannenbaum, Derek Wright, Karen Miller, and Miron Livny, Condor - A Distributed Job Scheduler, Beowulf Cluster Computing with Linux, The MIT Press, ISBN 0-262-69274-0, 2002.

[3] J. Frey, T. Tannenbaum, I. Foster, M. Livny, and S. Tuecke, Condor-G: A Computation Management Agent for Multi-Institutional Grids, Journal of Cluster Computing volume 5, 2002, pages 237-246.

[4] I. Foster and C. Kesselman, Globus: A Toolkit-Based Grid Architecture, The Grid: Blueprint for a new Computing Infrastructure, Morgan Kaufmann, ISBN 1558604758, 1999, p. 259-278.

[5] I. Foster, C. Kesselman, G. Tsudik, S. Tuecke. A Security Architecture for Computational Grids, ACM Conference on Computer and Communications Security, 1998, p. 83-92.

[6] K. Czajkowski et al., A Resource Management Architecture for Metacomputing Systems, Workshop on Job Scheduling Strategies for Parallel Processing, 1998, p. 62-82.

[7] J. Bester et al., GASS: A Data Movement and Access Service for Wide Area Computing Systems, Workshop on I/O in Parallel and Distributed Systems, 1999.

[8] William Allcock et al., Protocols and Services for Distributed Data-Intensive Science, Proceedings of ACAT, 2000, p. 161-163.

[9] S. Son and M. Livny, Recovering Internet Symmetry in Distributed Computing, Proceedings of CCGrid, 2003.

[10] D. Thain, S. Klous, and M. Livny, Deploying Complex Applications in Unfriendly Systems with Parrot, Journal of Supercomputing, 2004.

[11] P.F. Harrison and H.R. Quinn, Physics at an Asymmetric B Factory, The BaBar Physics Book, SLAC Report 504, 1998.