Resource Management for Interactive Jobs in a Grid Environment

Enol Fernández, Elisa Heymann, Miquel A. Senar
DACSO - Universitat Autònoma de Barcelona, Barcelona, Spain
enol@aomail.uab.es, {elisa.heymann, miquelangel.senar}@uab.es

Abstract

Most recent Grid middleware technologies have been aimed at the execution of sequential batch jobs. However, some users require interactive access when running jobs on Grid sites. Execution of these applications on a Grid environment is a challenging problem that requires the cooperation of several middleware tools and services. Additional problems arise when this interactive support is intended for parallel applications, which may run remotely across several sites. We provide transparent and reliable support for such applications. Our solution, based on the notion of split execution and interposition agents, allows running applications on a remote machine while some I/O operations are sent back to a home machine. The paper describes how we have applied interposition agents transparently to interactive applications and also describes a simple multi-programming mechanism that is used to start interactive applications as fast as possible even under heavy occupancy of resources. We provide a performance evaluation of the key elements involved in the execution of interactive jobs.

1. Introduction

Grid environments constitute one of the most promising computing infrastructures in computational science. One of their main goals is to tackle current and upcoming complex, large-scale problems by enabling the sharing of computational resources in a seamless way. Most of today’s grids are used for executing applications that follow a batch-processing approach. Typically, the user prepares a job, submits it to the grid, waits for it to be completed and retrieves the output after the job is executed. Nevertheless, there are currently increasingly more interactive applications that would benefit from being executed on a grid.

In the CrossGrid project [1], several applications that fall into the category of interactive applications have been developed and ported to a grid environment. These applications, which cover Medical, Environmental, and High Energy Physic (HEP) areas, allow the user to change parameters in near-real time. Therefore, support for remote steering of an application running on the Grid is required. The jobs are mainly MPI [2] distributed jobs, making interaction even more difficult.

The main requirements of the aforementioned interactive applications can be summarized as follows:

- Interactive Startup: the possibility of starting the application in the immediate future, also taking into account scenarios in which all computing resources might be running batch jobs.
- On-line Output Control: the ability to control application output online and to enable the user to decide whether to cancel this in accordance with the output results.
- Runtime Steering: this issue is similar to that described above, with an additional requirement being the ability to change the simulation-execution parameters while the application is running.

There are currently several mechanisms (such as SSH [3] or VNC [4]) that can be used to create interactive sessions between remote machines. Unfortunately, these mechanisms are not generally suitable for grid environments due to performance or administrative limitations. The use of VNC, for instance, introduces a substantial overhead, which results in a slow and sometimes unreliable graphics protocol when used in a wide area network. The standard distribution of SSH is not grid-enabled and does not support grid-related authentication. Even the use of an additional package [5], which adds support for the Grid Security Infrastructure, does not solve the problem because users do not have personal accounts on remote machines on which their jobs could run.

On the other hand, the existence of batch systems at each Grid site that have full control over local resources and jobs running on them imposes

---

1 This work was supported by the EU under contract RI-031857 and by the MEyC-Spain under contract TIN 2004-03388, and partially supported by the NATO under contract EST.EAP.CLG 981032.
significant restrictions on the fast startup of interactive jobs and significantly limits the number of actions that can be performed by a Grid brokering system to manage interactive jobs efficiently.

Taking the limitations demonstrated by existing mechanisms into account, we have developed several job-management services within the framework of the CrossGrid project that tackle the requirements of the aforementioned interactivity scenarios.

The paper is organized as follows: section 2 describes related work and section 3 presents the main components of the CrossGrid architecture and the middleware services involved in job execution. In section 4, we present the concept of split-execution and the way this has been applied to streaming application input/output. In section 5, we describe the specific mechanisms used in our system for scheduling interactive jobs. Section 6 presents a performance evaluation of the main components involved in the execution of interactive jobs. Finally, section 7 summarizes the main conclusions of this study.

2. Related Work

Research has begun in recent years into supporting interactivity on the grid. On the one hand, certain studies have explored the problem of interactive sessions and the redirecting of input/output data. In [6], an architecture for supporting interactive sessions is presented. The interactive desktop session links a remote execution node to the end-user’s submission node. The remote display solution used here is VNC [4], which is a useful but sometimes slow and unreliable graphics protocol. Glogin [7] provides an interactive shell while relying on Globus security. With Glogin, the user must first discover and select a remote site and manually establish the interactive shell to that site. Furthermore, some of its functionality requires privilege permissions on the remote machines. Another approach to interactive visualization is provided by the Sapphire plugin for the Unicore grid environment [8]. However, it is unclear at present whether the interactive visualization is applied at run-time or only in a post-mortem manner. Process steering has also been explored in previous studies that have considered independent jobs. Virtual Instrumentation [9] allows for the performing of iterative parameter adjustments in accordance with ad hoc heuristics for sequential jobs. It is unclear whether this mechanism could easily be extended to steer parallel jobs, as in our case.

On the other hand, our multi-programming mechanism bears similarities to the notion of Virtual Machines (VM) used as a means of multiplexing shared mainframe resources since the early 1970s. Some recent projects have focused on supporting unmodified applications (and their preferred environments) by using whole-system VMs running their own O/S. Oceano [10], COD [11], In_VIGO [12] and the XenoServer Open Platform [13], are related projects. In general, these require local clusters in which different VM configurations are available on a local server, these being instantiated upon request from remote-user applications. While these approaches provide maximum transparency for applications, VM instantiation incurs a non-negligible overhead and in some cases might require administrative access for machine execution. In our study, however, we emphasize virtualizing machines as a lightweight mechanism, with a very low overhead for controlling scheduling on remote machines. In addition, this could be applied to any remote site, regardless of the configuration adopted by the local administrator.

3. Overall CrossGrid Architecture

The architecture of the CrossGrid software defines different services [14], but we have restricted our discussion here to only those elements that are directly related to the execution of applications.

Each grid site is composed of a cluster of machines consisting of a gatekeeper and many worker nodes (WN) managed through a local queuing system, such as PBS or Condor. Figure 1 shows the main components involved in job management (a detailed description of which can be found in [15]). When users submit their applications, the component that is responsible for optimizing scheduling and node allocation decisions is our broker, called CrossBroker. CrossBroker obtains information on the status of each site through an information system built using Globus MDS [16]. It also performs the steps required to guarantee the effective submission of each job onto the selected resources, which may include different grid sites, as well as monitoring the application execution and reporting on job termination.

![Figure 1. Components of Job Management System.](image)

Jobs, which may be batch or interactive and, within each category, sequential, MPICH-P4 and MPICH-G2 [17] are submitted by specifying a job description
using the Job Description Language (JDL) [18], such as that shown in Figure 2.

The **Executable** and **Arguments** attributes specify the name and arguments of the application to be executed, while the **NodeNumber** attribute allows users to specify how many nodes their application will run on. The **JobType** attribute specifies the type of application (batch/interactive; sequential/parallel). The interactive parallel jobs currently supported include “mpich-p4” and “mpich-g2” jobs.

<table>
<thead>
<tr>
<th>Executable</th>
<th>&quot;interactive_mpich-g2_app&quot;;</th>
</tr>
</thead>
<tbody>
<tr>
<td>JobType</td>
<td>{&quot;interactive&quot;, &quot;mpich-g2&quot;};</td>
</tr>
<tr>
<td>NodeNumber</td>
<td>2;</td>
</tr>
<tr>
<td>Arguments</td>
<td>&quot;-n&quot;;</td>
</tr>
</tbody>
</table>

**Figure 2. Example of a job description using JDL.**

Our system includes a streaming facility and a multiprogramming mechanism which are specially intended to support interactive jobs. The user can include certain attributes in his job description file in order to specify how these mechanisms will be applied to her/his job. The **Streaming Mode** attribute can be:

1. **Reliable**: this streaming mode is intended for the execution of interactive jobs over unreliable networks and implies an intermediate buffering in a file of the I/O stream at both ends of the communication. If the input or the output fails to be sent, data will be written on the local disk. Regardless of why the input/output operation failed, our streaming mechanism will keep processes running and, at regular intervals, it will try the network connection again. If the connection succeeds, it will transfer any buffered data to the other communication end, and then resume normal operation.

2. **Fast**: this streaming mode is intended for execution of interactive jobs over reliable networks where the user is interested in considerably faster transfer. No intermediate buffering is performed in this mode, which means the data may be lost in case of network failure.

The **Machine Access** attribute controls the multiprogramming mechanisms and can be:

1. **Exclusive**: interactive jobs are executed on a machine that is idle and no multiprogramming components are used in this case. Response time is slightly longer in this mode, but the user’s application uses all of the memory and CPU of the remote machine without interference from our multiprogramming components.

2. **Shared**: interactive jobs are executed under control of our multiprogramming mechanism. This mode provides a faster startup but the interactive job may share the machine with a batch job. By using an additional attribute (Performance Loss), users may control how CPU time is distributed between its interactive job and the batch job. Values for Performance Loss can be 0, 5, 10, 15, and so on, and represent the percentage of CPU time the job leaves to the batch job.

In principle, the architecture of CrossGrid is similar to most existing Grid infrastructures, being batch-oriented submission system. However, our CrossBroker includes special features that support interactive-oriented job submission. In this study, we have only focused on these interactive features. Interested readers could refer to [15] to find more details about other features involved in the management of sequential and MPI jobs. Certain simple mechanisms deal with the allocation of resources when a new job is submitted to the system:

- **On-line scheduling**: The scheduler attempts to run each interactive job immediately. If the job enters a queue rather than immediately starting execution, it will be resubmitted to any other resource available.

- **Exclusive temporal access to resources**: This mechanism guarantees that a given resource is not matched to other applications for a certain period of time once the same resource has been allocated.

- **Randomized selection of resources**: This mechanism is used to generate different answers when there are multiple resource choices.

Obviously, the previous mechanisms are insufficient to fulfill all the requirements required by interactive jobs, which are even harder to achieve when the application is parallel and when tasks are executed concurrently on machines from different sites. We have included two additional mechanisms in our system:

- **I/O streaming**: Input has to be forwarded from the users (working on the submitting machine) and their application (running on different nodes of the grid). The output must follow the reverse path. Our proposed mechanism for supporting this requirement is explained in section 3.

- **Job multi-programming**: This mechanism is used when no free resources are available at submission time and we want to start the execution of the application shortly after submission. Our proposed mechanisms are explained in section 4.

### 4. Support for I/O streaming

In this section, we tackle the problem of forwarding the input of a process from the submitting to the executing machine, and the output following the reverse path, in such a way that users get the sensation...
that their job is being executed locally. Although applications (both MPICH and sequential) are executed on remote sites, the input/output of such applications should be controlled locally, that is, from the submitting machine. That way, users can interact with their applications that are executing remotely.

Our system uses mechanism or interposition agents [19] and applies them transparently to both sequential and parallel jobs. Users do not need to recompile their application before submitting it to our environment. They will only need to specify the value of certain attributes in the job description (see Figure 2).

Interactive sessions are handled by a Grid Console (GC) to obtain mostly-continuous input/output from remote programs running on an unreliable network. A GC is a split execution system made up of two software components: an agent (Console Agent – CA) and a shadow (Console Shadow – CS or Job Shadow – JS).

Figure 3 shows the simplified schema relating the different CrossGrid components and the agents for supporting interactivity. As in the non-interactive case, users submit their interactive application through the User-Interface (UI) Command Line. The jdl file is submitted to the CrossBroker, which, in turn, prepares the application-execution environment; following this, the MPICH subjobs are submitted to the different gatekeepers, from where they are subsequently distributed among the Worker Nodes (WN) for execution. The dotted line in Figure 3 shows the interactivity path. Standard input messages go from the Job Shadow to the Console Agent, which then forwards these to a subjob, while both standard output and standard error messages follow the reverse path. The CA traps some of the procedure calls for an application, and forwards them (via RPC) to a shadow process (CS) on another machine (listening in a randomly selected port probing for an available port or manually specified by the user in an attribute field of the job description file; this is required if, for instance, users want to use a predefined port that is open in their local firewall). Under this arrangement, a program can run on any networked machine and still execute exactly as if it were running on the same machine as the shadow. All the network communications are GSI-enabled and are therefore a secure connection.

The Console Agent runs on a Worker Node and consists of a shared library that intercepts reading and writing operations on stdin, stdout, and stderr of the running job. When possible, the CA sends the output back to the CS. The shadow manages the input and output streams in accordance with the agent’s request and the streaming mode specified by the user in the job submission file. When the reliable mode is selected, both the CA and the CS write data to the local disk and retry failed operations at regular intervals for a certain number of times, after which they will give up and kill the process. The number of retries and the number of seconds between each retry are configurable.

In our environment, in the cases of either sequential or MPICH-P4 applications, one CA is executed, while in the case of MPICH-G2 applications, multiple CAs are executed. Figure 4 illustrates a schematic representation of the CAs and CS involved when executing an MPICH-G2 application enabling bidirectional communication. In this case, every subjob can perform output. The input will be forwarded to every subjob and it is the users’ responsibility to guarantee that input will be read by a single subjob, for example subjob 0 (this is done by checking the MPI rank value of each task). The console shadow is executed on the UI machine. All of the job’s tasks have both an output and an input stream connected to the job shadow.

**Figure 3. Interactivity management in CrossGrid.**

It is worth noting that buffers have been included in both the submitting and executing machines to provide users with a genuine feeling of interactivity. In the case of MPICH-G2 applications there is an output buffer on each executing machine, each collecting the output of the corresponding MPI subjob. The content of this output buffer is sent to the output buffer associated to the Job-Shadow process located on the submitting machine, which in turn is flushed to the screen. This flushing is produced in 3 cases:

- When the output buffer on the user machine is full.
- When a timeout occurs.
- When an “end of line” is found.

The input provided by users on the submitting machine is forwarded to the input buffers associated to each Console Agent on the executing machines. The forwarding is produced when the “enter” key is hit. This flushing is intended to provide a readable output for users, and a “natural” appearance.
5. Resource Allocation for Interactive Jobs

There are two main issues that control the way in which resources are allocated to jobs: user priorities and multi-programming. We describe these below.

5.1. Basic Resource Allocation Scheme

For assigning resources to applications, our Job-Management System includes an accounting mechanism that is used to implement a fair-share priority algorithm [20]. Users are associated a dynamic priority, which determines how many resources they can use at a given time. User priorities are computed according to the following expression:

\[ P(u, t) = \beta \cdot P(u, t-\delta t) + (1-\beta) \cdot a_f \cdot r(u, t) \]  

(1)

Where \( P(u, t) \) is the priority of user \( u \) at time \( t \) (the higher the value of \( P \), the lower the priority it represents), \( r(u, t) \) is a normalized value representing the number of resources used by user \( u \) at time \( t \), \( a_f \) is the application factor described below, and \( \beta = 0.5 \cdot \delta h/h \) is \( h \) the half-life period (the rate at which the priority value improves). User priority is updated in accordance with the previous formula, also taking into account the characteristics of their job, which are reflected in the value of the application factor \( (a_f) \):

- Batch jobs worsen the priority according to the resources used. This corresponds to having \( a_f = 1 \).
- Interactive jobs worsen the priority faster than in the previous case. In this case, \( a_f = 2 \cdot \text{Performance Loss/100} \), as specified in the job attribute.
- Batch jobs already in execution may be forced to yield their machine to execute an interactive application by means of the multi-programming mechanism described below. In this case, \( a_f = \text{Performance Loss/100 of the interactive application} \) (the priority corresponding to its user will be worsened to a lesser extent than in previous cases).
- Priorities are dynamically updated, so if users do not use any resources at all, the original number of credits will gradually be restored, according to \( h \).

User priorities are updated every \( \delta t \) times for each user whose current priority is different (worse) than the initial priority.

By using this user-priority scheme, we prevent users from always submitting their jobs as “interactive” and therefore saturating the system, preventing real interactive jobs from being executed. If there are not enough available resources, jobs belonging to users with worse priority are rejected.

Ideally, interactive applications should always run immediately after submission. However, there may be situations in which remote resources are not available because they are running batch jobs. We describe below a multi-programming mechanism which gains control of the remote machine in such a way that enables the existing batch job to stop without losing this machine. The interactive job will start there and share the machine with the already-running batch job.

5.2. Multi-programming Jobs

Our job management service has a multi-programming mechanism that enables both interactive and batch jobs to share a single machine, in such a way that the interactive application starts its execution as soon as it is submitted (unless all resources in the Grid are busy executing other interactive applications). This multi-programming scheme takes advantage of the Condor Glide-In [21] mechanism, and is based on the transparent submission of job agents for jobs submitted by the user. The agent gains control of remote machines independently of the local-site job manager.

Each machine acquired by our agent is configured as two virtual machines, in order to create a separate group of dedicated resources for two types of application: batch, on the one hand, and interactive, on the other. It is worth noting that our concept of virtual machines is lightweight and does not correspond to the classic view of virtual machines [22] that presents the image of multiple Operating System configurations (completely isolated from each other) sharing a single machine. In our case, the machine only runs one O/S, but we split the machine into two separate execution slots. Each slot contains the executable and files required by the corresponding job. From a logical point of view, batch jobs will run on one virtual machine and interactive jobs will run on the other. However, our agent guarantees that interactive jobs will be executed at a higher priority than batch jobs. When the interactive job is finished, the original priority of the batch job is restored and after completion of the batch job, the agent leaves the machine.

Figure 5 illustrates all possible scenarios that CrossBroker deals with:
1. Sequential Batch Job Submission. This submission triggers the execution of an agent if there is either an available machine or space in the queues managed by the local scheduler. Once started, the agent will create two virtual machines on the Worker Node/s: one for batch jobs (batch-vm) and another for interactive jobs (interactive-vm). The batch job will start its execution on the virtual machine devoted to batch applications (batch-vm). This situation is illustrated by arrow (1) in Figure 5. Special care has to be taken if the agent is killed (by the local scheduler, by failure of the machine it is running on, etc.). In this case, new agents will be submitted when possible.

2. Interactive Application Submission (exclusive access mode): CrossBroker tries to allocate a free machine. If there are CPUs available that meet the job requirements, the job will be submitted to it without any agent. (Illustrated by arrow (3)).

3. Interactive Application Submission (shared access mode): CrossBroker first searches for machines with agents, using the available interactive virtual machine (interactive-vm). The job will be sent to one of the virtual machines, which immediately meets the job requirements, causing the batch job executing on the other virtual machine to lower its priority so as to benefit the interactive job (according to the Performance Loss specified by the interactive job). This is illustrated by arrow (4) in Figure 5. If no free interactive agents are found, CrossBroker searches for an idle machine and submits the agent and the application in a similar way to it does in the case of a batch job.

If there are no idle machines or there is no space in the local scheduler’s queues, batch applications are queued in the CrossBroker to wait for a machine to become idle. This situation is illustrated by arrow (2) in Figure 5. However, if there are not enough machines (with or without agents) to execute an interactive application, its submission will fail. An interactive application will never pre-empt another already-running interactive application.

Although not shown in Figure 5, it is possible to have a combination of machines with and without agents for executing a parallel interactive application.

The agent-based mechanism improves resource availability for interactive jobs that will even be able to run under the circumstances of high Grid-resource occupancy. On the other hand, this has little impact on batch jobs that will undergo some execution delay when sharing their CPU with interactive jobs. However, given the nature of batch jobs, this delay is not particularly problematic and its impact is compensated for by the more moderate worsening of user priority.

![Figure 5. Multi-programmed execution of jobs.](image)

Obviously, the OS overhead of multi-programming incurs a certain cost, but assuming there is adequate physical memory for all running jobs, this should be minimal. In fact, our multi-programming system could allow a larger degree of multi-programming, creating dynamically more than two virtual machines. We are actively investigating several issues that may affect the management of user priorities and virtual machines, also taking into account the behavior of applications.

6. Implementation and Performance Measurement

A complete prototype of CrossBroker has been included as part of the CrossGrid middleware and used in the project testbed. The testbed is composed of 18 sites in nine countries. The current basic middleware is based on Linux, Globus 2.4.x [16] and certain external tools taken from the DataGrid project [23]. Most sites offer storage capacities above 600GB. The hardware type ranges mostly from Pentium III to Pentium Xeon based systems, with RAM memories up to 2GB.

In the following subsections, we describe some experiments that we conducted to measure the performance of the interactive services described above and the overheads incurred by them. From the point of view of an interactive application, key system metrics include the response time (ideally, interactive application should start soon after submission with minimum delay) and the latency of I/O streams (so that the user gets a similar feeling to when the application is running locally). These two elements are analyzed, respectively, in the first two subsections. The last subsection analyzes the overhead introduced by our multi-programming mechanism by taking into account the performance loss that it introduces to interactive jobs.

In the tests described in subsections one and two, we used two scenarios: the first consisted of a campus
grid, where the submission and the execution machine were connected by our campus university network (100Mbps). In the second scenario, the client machine was located in our department and the execution machine was located at the IFCA center in Santander (both centers are connected by the Spanish Internet network). In all cases, CrossBroker was running on one of our department’s machines.

6.1. Response Time Evaluation

Similarly to other brokering systems, CrossBroker automatically performs several steps once a new job is submitted to the system:

- Resource discovery: this phase gathers information on the available resources at the time of submission. The time spent depends mainly on the bandwidth and latency between the CrossBroker and the information system (in our case is located in Germany, while the CrossBroker is in Spain). This phase takes around 0.5 seconds.

- Selection of the best resource. Information obtained in the first step is used to initially filter sites that do not fulfill all job requirements (for instance, in terms of CPU architecture and OS version). From the list of resultant sites, information may not be completely accurate and, therefore, CrossBroker contacts each remote site individually and gets the most updated information about the state of their local queues. This phase depends on the number of discovered sites and the user’s requirements. With no special requirements and a set of 20 remote sites, located all over Europe, the CrossBroker spends around 3 seconds on this selection.

- Job submission to execution at remote site. In CrossBroker, this step also involves a series of additional actions to prepare automatic transfer of input files that must be moved to the remote site.

It is worth mentioning that the first two steps mentioned above are not required for interactive jobs that want to run on an Interactive Virtual Machine because the information about existing VMs is kept locally by CrossBroker.

We have measured different cases of submission of a sequential job using CrossBroker and we also measured job submission using Glogin (described in section 2, for which we set up special machines due to some of its privilege permission requirements on the remote machines). We submitted 100 jobs in each case and average results are summarized in Table I, where we show the time spent in each step from when the jobs enters CrossBroker until it starts execution. Times for resource discovery and selection steps are applicable only when a batch job or an interactive job in exclusive mode are submitted. For Glogin, the user must select the destination machine manually and be sure of its availability before executing the application. Interactive jobs in shared mode use a combined step inside CrossBroker for discovery and selection. The submission columns of Table I correspond to the response time, that is, the time elapsed between the instant when the job is finally submitted, either by Glogin or by CrossBroker, to the remote gatekeeper and the instant when the first output arrives in the user machine (i.e., the time includes the traversal of Globus layers and the local queue system at the remote site).

Table I. Response time for jobs (seconds).

<table>
<thead>
<tr>
<th>Method</th>
<th>Resource Discovery</th>
<th>Resource Selection</th>
<th>Submission</th>
</tr>
</thead>
<tbody>
<tr>
<td></td>
<td>Campus Grid</td>
<td>IFCA</td>
<td></td>
</tr>
<tr>
<td>Glogin</td>
<td>Hand-made by user</td>
<td>16.43</td>
<td>20.12</td>
</tr>
<tr>
<td>Idle</td>
<td>0.5</td>
<td>3</td>
<td>17.2</td>
</tr>
<tr>
<td>Virtual machine</td>
<td>0.5</td>
<td>3</td>
<td>6.79</td>
</tr>
<tr>
<td>Job + agent</td>
<td>0.5</td>
<td>3</td>
<td>29.3</td>
</tr>
</tbody>
</table>

As shown in Table I, submission of interactive jobs in shared mode exhibits the shortest startup times. It is more than two times smaller than the best of the other options (Glogin) in all steps. The existence of a direct communication between CrossBroker and all the remote agents is the reason for these good results. This communication makes resource selection more efficient and avoids the overhead incurred by other middleware layers on remote machines (Globus middleware and local batch systems). Glogin submission and interactive submission in exclusive mode exhibit similar performance, although Glogin is slightly better. CrossBroker actually performs some extra actions compared to Glogin in order to prepare automatic staging of job input files and it also uses a two phase commit protocol that guarantees a better detection of error conditions at submission time. As expected, the worst time corresponds to the submission of a batch job because, in addition to the user job, this time includes the transfer and execution of our agent on the worker node. This extra overhead normally only affects batch jobs, which are supposed to run for several hours, and hence can be considered negligible.

6.2. Sequential IO Streaming Overhead

To measure the performance of the IO streaming, we ran an experiment over the two aforementioned grids. Using a test suite written by ourselves, we measured data transfer times between a submission machine and an execution machine. A client and a server process were created in the submission and execution machines, respectively. The client and server executed a coordinated sequence of 1,000 read/write
operations to their stdin and stdout. A single sequence consisted of the client writing data to its stdout, data was read by the server in its stdin, and then the server wrote an answer in its stdout that was finally read by the client in its stdin. Data transferred in each read/write operation varied from 10 bytes to 10K, and we measured the round trip incurred by each sequence of read/write operations executed by the client process.

We compared the data transfer times incurred by three different mechanisms:

- ssh: we established a regular ssh session between the submission machine and the execution machine and we started the client and server processes manually. It is worth mentioning that this mechanism is commonly used in local area networks but is not available, in general, in a grid due to restrictions imposed on remote machines.

- Glogin: as explained earlier, provides an interactive shell while relying on Globus security.

- Interposition agents: this is our method that was used out-of-the-box, without any special set up required either for the submission machine or the execution machine. We tested interposition agents using both the reliable and the fast transfer modes, as explained in previous sections.

The same experiment was executed using a client machine in our department and the server machine running on a machine in our campus grid or in Santander. Results of both scenarios are shown in Figure 6 and Figure 7, respectively, which show the data transfer time of each read/write sequence for 10 and 10K bytes transfers. The X-axis represents each sequence and the Y-axis shows times in seconds for each sequence.

In general, our fast transfer mode achieves very good performance in all cases. It is the method that exhibits the best transfer times when machines were located in the campus grid. In the wide area grid, its results are similar to those achieved by ssh and Glogin for data transfers sized between 10 and 1K bytes. However, our method exhibits a higher variance.

Glogin does not perform very well in the campus grid or for large sized data transfers (10K bytes) in the wide area grid. Data transfers using interposition agents in a reliable mode are usually the slowest
method due to the extra overhead incurred in disk write and read operations. Obviously, those extra operations are the price paid for more reliability in data transfers, a reliability that is not available in the other methods. Curiously, our reliable method performs very well for large data transfers (it is better than ssh in a campus grid and similar to ssh in the wide area grid). We believe that this good behavior can be explained by the fact that, compared to ssh, our method uses larger internal buffers. Therefore, the disk overhead is compensated by a smaller number of IO operations.

6.3. VM Load Overhead

Finally, we evaluated the overhead incurred by our multiprogramming schema based on lightweight virtual machines. We will also show that we can quite accurately control the percentage CPU time that the interactive jobs gives to a batch job according to the value included in the Performance Loss attribute. In order to show the impact of the execution of an interactive job in a virtual machine sharing resources with a batch job, we wrote an interactive job which iterates 1,000 times. At each iteration, the application performs an I/O operation followed by a CPU burst. We measured the time elapsed during each of these operations in three different cases:
- Execution in exclusive mode: the job runs alone on an idle machine. This case gives us the base line to compare with the remaining cases.
- Execution in shared mode alone: the job runs on an interactive VM; no batch job runs on the batch VM.
- Execution in shared mode: an interactive and a batch job run simultaneously on the interactive and batch VMs, respectively. We tested execution with two Performance Loss values (10 and 25).

Figure 8 shows the execution time of CPU bursts (left) and data transfer times (right) of each iteration. The X-axis represents each loop iteration and the Y-axis shows times in seconds for each operation. First, the overhead introduced by our multiprogramming agent is negligible both in terms of CPU and I/O. The times obtained by the job running in exclusive mode and the job running in shared mode alone are nearly the same. Both curves are indistinguishable in the two graphics in Figure 8.

Second, Figure 8 also shows that our system exhibits highly accurate control of CPU time distribution between the interactive job and the batch job when both run in shared mode.

![Figure 8. VM overhead. (left) CPU overhead and (right) I/O overhead.](image)

The interactive job running alone constitutes the reference execution; it has a mean CPU time of 0.921 seconds and 0.00606 for I/O time, and a standard deviation of 0.001 and 6.9e-5 respectively. In the case of a Performance Loss equal to 10, mean CPU time is 1.004 seconds with a standard deviation of 0.004 (8% worse than the reference case), and for I/O, the average time is 0.00632 seconds with a standard deviation of 8.0e-5 (5% worse than reference execution). In the case of a Performance Loss equal to 25, mean CPU time is 1.132 seconds (standard deviation of 0.010) and average I/O time is 0.00661 seconds (standard deviation of 7.0e-5). These values correspond to performance losses of 22% and 10% respectively.

To summarize, we can conclude that CPU adjustment is close to the value of the Performance Loss attribute, while the priority adjustment has a lower repercussion on I/O performance because it mainly depends on the network bandwidth.

7. Conclusions and Future Work

Traditionally, many scientific applications submitted to a Grid are executed in a pure batch-mode. Applications run unattended in the background and, consequently, are not able to perform interactive input and output. Adding interactive execution capabilities to the grid middleware was one of the main goals of the
CrossGrid project. Under this assumption, users should be able to run their application on remote grid resources while being able to simultaneously interact with the application by performing interactive input, receiving the output stream as it is produced by the application. Implementing such a system is no simple matter and may result in a solution that reveals many implementation details to users.

We have developed a solution to support interactive application on a Grid in which no changes to user code are required. Our tools are based on the idea of split execution systems, in which an agent is placed between the application and the operating system and traps some of its input/output system calls. The agent routes the trapped calls to a shadow process that executes these calls on the home machine. With this simple execution model, we are able to run existing, untouched, executable programs and hide all of the implementation details. We have applied interposition agents to sequential and MPI applications, incurring a minimal overhead for the whole system.

We also include certain specific features in the scheduler. In particular, a simple multiprogramming mechanism enables the execution of interactive applications when no free resources are available or when a very fast start-up time is required.

The overall performance obtained by our streaming solution is highly competitive in comparison with other existing mechanisms such as ssh or Glogin, which, in contrast to our system, exhibit certain limitations when applied to a Grid and are only applicable to sequential jobs. Our system also includes a reliable transfer mechanism that can be selected to run applications even in the presence of temporal network failures.

The overhead of our multiprogramming mechanism proved to be negligible. It provides interactive applications with a highly accurate system that enables the controlled distribution of CPU between interactive and batch jobs according to user preferences.

There are other aspects of our infrastructure that suggest the need for further research, including priority management and control of the degree of multiprogramming, so as to dynamically adapt this to the behavior of different types of interactive applications; transparent streaming of other IO traffic; and tunneling capabilities through firewalls without a range of available ports open for Globus.

8. References


