Extensible Job Managers for Grid Computing

Paul D. Coddington Lici Lu Darren Webb Andrew L. Wendelborn

Department of Computer Science
University of Adelaide
Adelaide, SA 5005, Australia
Email: {paulc,andrew,darren}@cs.adelaide.edu.au

Abstract

Grid computing is becoming an important framework for enabling applications to utilize widely distributed collections of computational and data resources, however current grid software is still immature and rather difficult to use. The Globus Grid Toolkit is a set of low-level tools, protocols and services that has become a defacto standard for basic grid computing infrastructure. The Globus Resource Allocation and Management (GRAM) service provides for the management and remote execution of jobs defined using a standard Resource Specification Language (RSL). Currently, the GRAM has very limited functionality, which makes it more difficult to develop grid applications. One limitation is the lack of support for applications that require a special execution environment, such as Java applications that run within a Java Virtual Machine. Cumbersome workarounds are necessary to run such applications. The current GRAM addresses these problems in a rather ad hoc way for certain specific cases, however there is no general, well-defined mechanism for supporting arbitrary execution environments. Here we outline some of the problems with the current Globus GRAM specification and provide a proposal for how they might be addressed by defining some extensions to the standard RSL supported by the GRAM, as well as some modifications to the design of the GRAM that would enable it to support arbitrary execution environments. We give examples of how our proposed system can provide improved support for Java applications and cluster management systems, and describe our ongoing work in implementing prototypes of these proposed GRAM extensions.

Keywords: Grid computing, Java, cluster management systems.

1 Introduction

A grid computing environment is one in which applications can utilize multiple computational resources that may be distributed at widespread geographic locations. Each resource is typically a high-performance computer or a cluster of workstations, however the term encompasses other resources including storage facilities (e.g. disk arrays or tape silos), data archives (e.g. a repository of satellite data), data visualization (e.g. virtual reality environments), and even computer-controlled scientific instruments (e.g. an electron microscope). Computational grids are seen by many as the next step in the evolution of internet technologies, providing "dependable, consistent, pervasive and inexpensive access to high-end computational capabilities" (Foster & Kesselman 1999) for a variety of end users.

Copyright ©2003, Australian Computer Society, Inc. This paper appeared at the Twenty-Sixth Australasian Computer Science Conference (ACSC2003), Adelaide, Australia. Conferences in Research and Practice in Information Technology, Vol. 16. Michael Oudshoorn, Ed. Reproduction for academic, not-for profit purposes permitted provided this text is included.

Despite its potential, grid computing is still an immature technology that is not widely used. Current grid computing software is rather low-level, immature, difficult to set up and use, and requires substantial technical expertise. Currently, most grid computing applications are in the areas of computational science and engineering that have traditionally been major users of high-performance computing. A much wider variety of applications is likely once grid computing technologies become easier to use and more sophisticated.

The Globus Grid Toolkit (Foster & Kesselman 1997) is a set of low-level tools, protocols and services that has become a defacto standard for basic grid computing infrastructure. A fundamental aspect of any grid computing system is the mechanism for the specification, management and remote execution of jobs. The Globus Resource Allocation and Management (GRAM) service is run on every node of a Globus grid. The GRAM runs a jobmanager service that is responsible for job execution and monitoring, and accepts jobs defined using a standard Resource Specification Language (RSL).

Currently, the GRAM has very limited functionality, which makes it more difficult to develop grid applications. The job definition using RSL allows the user to specify the executable file and redirection of standard input and output to named files. All these files are copied to and from the remote execution host by the jobmanager. However, unlike most other systems for remote job execution (such as Condor, PBS, Nimrod and other cluster management systems) there is no mechanism for specifying other files (e.g. input and output data files) that must be copied to and from the remote execution host. This file staging has to be done separately by the user, whereas this could more easily be handled by the jobmanager.

Another problem with the GRAM occurs when the executable is not run directly, but requires a special execution environment. For example, Java applications are run on a Java Virtual Machine (JVM) by invoking the `java` command followed by the name of the executable. An interpreter is required to run programs written in languages such as Perl and Python. Parallel programs written in MPI are executed using a job loader (e.g. the `mpirun` command for the MPICH implementation of MPI) that takes the number of processors to be used as a parameter. The standard GRAM jobmanager does not understand these execution environments and would try to run the executable file directly. Cumbersome workarounds are required to run such programs using the current GRAM.

A similar problem occurs in executing jobs using cluster management systems (CMS) such as Condor,

PBS and Nimrod, which schedule, run and monitor a job (or a set of jobs) across a cluster or network of computers. Again, the executable is not run directly on a machine, but rather is passed to a submission program (e.g. `condor_submit` for Condor and `qsub` for PBS) that typically accepts a shell script or a job specification file describing the program to be executed. It is not possible to submit a CMS job (e.g. a Condor or PBS job submission file) directly to the GRAM, although there are some grid-enabled systems such as Condor/G (Condor Project 2002) and Nimrod/G (Abramson, Giddy & Kotler 2000) that provide a CMS-like interface. Typically the CMS job must be first converted to standard RSL for submission to the GRAM, and may then converted back to a CMS job format for submission on the remote execution host. Unfortunately some useful information may be lost in these conversions, since the basic RSL fields understood by the GRAM are quite limited compared to the job specification syntax for most cluster management systems.

The current implementation of the GRAM has some mechanisms to alleviate these problems in certain specific cases, currently MPI and Condor jobs. However these seem rather ad hoc, and there is no general, documented, well-defined mechanism for adding support for additional execution environments to the GRAM. These limitations of the GRAM make it more difficult to develop grid applications that require special execution environments (such as Java programs), or to use cluster management systems that do not have special support in the GRAM (i.e. all except Condor).

This paper provides a discussion of some of the problems with the current Globus GRAM specification and a proposal for how they might be addressed by defining some extensions to the standard RSL supported by the GRAM, as well as some modifications to the design of the jobmanager that would enable it to support arbitrary execution environments. Section 2 of the paper gives a brief overview of the relevant aspects of the Globus Grid Toolkit. Our proposal for a more general, extensible software architecture for a grid jobmanager is outlined in section 3. We give examples of specific instances of this architecture for supporting Java applications (in section 4) and cluster management systems (in section 5), and describe our ongoing work in implementing prototypes of these systems.

2 The Globus Grid Toolkit

The Globus Toolkit is the defacto standard for grid computing. The Toolkit is deployed in large testbeds and used to implement a variety of applications. It provides a simple, well-defined interface to a wide range of services supported on a highly heterogeneous mix of systems.

In this section, we introduce the Globus Toolkit, its architecture and its components. We focus on the Toolkit component for executing and managing jobs on remote compute resources and discuss some problems that arise in trying to utilise this component, particularly for applications requiring special execution environments.

2.1 Globus Toolkit

The Globus Toolkit is a "bag of services"; a set of components from which a developer can select to meet their needs. The components include basic services for security, information, resource management, storage and communication. Each service is distinct and has well-defined interfaces so they can be incorporated into applications or tools in an incremental fashion.

The layered architecture of the Toolkit is analagous to an hourglass. At the neck of the hourglass is a small set of core abstractions and protocols from which many different high-level behaviours can be mapped onto many different underlying technologies. These abstractions and protocols provide uniform access to diverse implementations of local services, and building blocks upon which global services can be built. A local site need only provide these local services. Global services can be built without knowledge of local implementation.

Among the main components of the Globus Toolkit are the Grid Security Infrastructure (GSI), Grid Information Service (GIS), Globus Resource Allocation and Management (GRAM), and the Globus Access to Secondary Storage (GASS). The GSI enables secure authentication and communication on the grid. The infrastructure is based on public key encryption with a "single sign-on" model for delegating credentials between resources. The GIS provides the information infrastructure. Also called the Monitoring and Discovery Service (MDS), it provides a unified means for querying resource information and constructing namespaces for resource information involving many organizations. The GRAM is the resource management component. It provides a standard interface to local job scheduler systems. GASS provides a range of services for secure remote file access and transfer.

One role of the GRAM is enabling the execution and management of jobs on remote compute resources independent of the local resource management system. Next, we describe the GRAM component and in particular its method for introducing new local scheduler systems.

2.2 Globus GRAM and the Jobmanager

The GRAM is an integral component of a computational grid. It is responsible for processing job requests, enabling remote monitoring of jobs, and updating the information services with information regarding the resources it manages.

The GRAM component provides a standard interface for computational grid tools and applications to express resource allocation and management requests. The interface enables individual sites to independently select their choice of scheduler. The GRAM currently operates in conjunction with a number of schedulers including NQE, EASY-LL, LSF, LoadLeveler, Condor, SGE, PBS, and a simple "fork" scheduler. In addition, schedulers including Condor/G and Nimrod/G map their respective interfaces to the GRAM interface. We discuss these further in Section 5.

The GRAM fulfils resource requests specified with the Resource Specification Language (RSL). The RSL is the language that a client uses, among other things, to describe a job and its requirements. RSL is based on the LDAP query language with various standard fields. These fields include the executable program, arguments, standard input and standard output. In addition, the RSL allows a client to specify a remote location for the executable program, standard input and standard output. If a remote location is specified, the GRAM uses GASS to perform a remote copy. The following expression illustrates a job request in RSL:

```
&(count=1)
 (executable=/bin/echo)
 (arguments="Hello Globus World")
```

This expression requests one execution of the specified program with the provided arguments. RSL also provides fields for setting the environment, setting the job type, and others. The GRAM maps these standard fields to arguments for a particular scheduler.

The major GRAM components are the Gatekeeper and the Job Manager.

The Gatekeeper is responsible for accepting allocation requests, mutually authenticating with the client, mapping the requester to a local user, starting a job manager as the local user, and passing the allocation arguments for processing.

The Job Manager is created by the Gatekeeper to process an allocation request. It parses the RSL, and maps the request to a local scheduler. The Job Manager then handles all further communication with the client. It maps GRAM job management operations (e.g. poll and cancel) to scheduler operations, and scheduler callbacks (e.g. job request reply and status change) to GRAM callbacks. The mapping is provided by a Scheduler-Specific Plugin.

Each Scheduler-Specific Plugin maps Job Manager functionality to a local scheduler. A Plugin implements a small set of operations common to most scheduling systems. Each Plugin provides a set of programs that map Job Manager requests to scheduler commands. The set of programs include job queue, submit, poll and remove. The queue script interprets input from the Job Manager and invokes the equivalent scheduler command. The client accepts a default Plugin, or explicitly specifies the Plugin (e.g. to use the Condor Plugin, the client specifies `jobmanager-condor`).

The Job Manager provides the Plugin architecture for extensibility. The introduction of a new scheduler requires only the implementation of the relevant scripts. However, the Plugin architecture is not applicable in all situations. For example, a parallel program written in MPI requires a special execution environment in the form of a loader program (such as the `mpirun` program for the MPICH implementation of MPI) that loads the executable onto all the processors and initiates the job. Unlike a scheduler, which works with any job, the MPI loader is only relevant to MPI jobs, so the Plugin architecture is not applicable. However the standard GRAM jobmanager would not understand how to run this kind of job, which requires a special execution environment.

In order to handle this situation, Globus provides an additional RSL field called called `jobType`, in order to tell the Job Manager that the job requires a special execution environment. Setting `jobType=mpi` indicates the Job Manager needs to start the specified executable with the MPI loader program. Of course, this requires some additional code in the Job Manager to understand the `mpi` job type and execute it in the desired fashion, by instantiating the MPI loader program with the appropriate parameters. Unfortunately this capability is currently limited to MPI jobs that are submitted using a Globus-enabled version of MPICH. It is not evident how to extend this to the general case, in order to avoid these problems in other execution environments.

2.3 Globus and Java

Similar problems are encountered in using Java programs with the GRAM, since Java applications also require a special execution environment – the Java Virtual Machine (JVM). However unlike MPI, there is no special job type defined for Java that is understood by the GRAM.

A Java program is executed by providing the JVM program (`java`) with parameters such as the location of classes and the amount of memory to be allocated to the JVM; the name of the class containing a `main()` method; and program arguments. For example, a request to execute a Java program named `Main` whose class is stored in a JAR file named `classes.jar`, parameter `name=value` and program argument `arg1` can be specified with the following RSL statement:

```
&(executable=java)
 (arguments=-classpath classes.jar -Dname=value \
            Main arg1)
```

There are several problems with this RSL request. Clearly the syntax is not intuitive – the `executable` attribute specifies the Java Virtual Machine program (i.e. the execution environment rather than the executable for the application program) and the `arguments` attribute is littered with executable details. The actual program executable is the `Main` class and its codebase in `classes.jar`. The arguments provided are a mixture of parameters to be passed to the JVM (the execution environment) and arguments for the program. Also, the programmer needs to provide workarounds to enable the program to execute. By default the GRAM will stage the JVM executable file and not the JAR file containing the program executable, which is the opposite of what we want (this problem is discussed in more detail in the next subsection). There are also other issues, particularly if the application requires a specific version of Java.

Globus provides support for Java through the Java Commodity Grid (CoG) Toolkit and the Java GRAM (von Laszewski, Foster, Gawor & Lane 2001). The Java CoG provides clients with a Java binding to the Globus service APIs. The Java GRAM provides an implementation of the GRAM service written in Java. In addition, the Java GRAM forks a virtual machine if the executable argument is a JAR file, which addresses many of the limitations described above. However, using the Java GRAM requires the replacement of the standard GRAM and does not generalize to other execution environments.

2.4 Problems with the GRAM

There are several problems and limitations with the current Globus GRAM implementation. A major limitation is that the standard RSL fields understood by the GRAM lack basic functionality such as specifying files (other than the executable, and redirections of standard input and standard output) for staging to and from the remote execution host. The programmer has to provide this file staging.

The main issue we are attempting to address is the lack of support by the GRAM for arbitrary execution environments. As we have demonstrated in the case of a Java application, specifying a job that requires a special execution environment requires mangling the semantics of a standard RSL request. The executable field must be used to specify the program to be run in order to instantiate the execution environment (e.g. the Java Virtual Machine or the Perl interpreter), and the arguments field is used to specify parameters for the execution environment, the actual program executable to be run, and the arguments to this program.

Apart from the confusing and non-intuitive syntax of this request, a complication arises when using this approach. By default the GRAM will automatically copy the file specified as the executable to the remote execution host, however we would need to override this action since the software for the execution environment (e.g. Java or Perl) should already be installed on the remote machine. The GRAM has no

way of telling what is the actual executable file that does need to be copied to the execution host, and unfortunately there is no way of even telling the GRAM to copy the actual executable file, since it does not support the specification of files for staging. So an added burden is placed on the programmer, to use GASS to copy the executable to the execution host.

The GRAM does support certain execution environments, but in a rather ad hoc fashion. The `jobType` field in the standard GRAM RSL currently supports only two special types of job, MPI and Condor. The Java environment is only directly supported by using a Java version of the GRAM. Ideally there would be a more general, well-defined and easily extensible mechanism for supporting any execution environment. This work explores an approach for providing such a mechanism.

One of the goals of grid computing is to support submission of jobs to a grid of multiple compute clusters, either in a local area (a "campus grid") or a wide area (e.g. a national or global grid). These clusters will all be running their own cluster management system (CMS), which may not all be the same. The GRAM provides a very useful (and necessary) mechanism for translating a GRAM job specification in RSL into a job submission script for a variety of CMSs, using the scheduler-specific plugin architecture. This is straightforward since the RSL job specification is restricted to very few fields, basically just those that would be used by the default fork jobmanager. Unfortunately this means that information that could be useful for the CMS scheduler on a cluster compute resource is not available.

The main issue here is that the GRAM (and the `globusrun` program used to submit jobs to the gram) assume that an appropriate computational resource has already been identified by a resource broker or scheduler based on some resource requirements for the application (e.g. architecture, operating system, memory, software availability), so the job is submitted to a particular node in the grid that satisfies these requirements. Hence the standard RSL job submission request only provides job execution details, not resource requirements. This is different to a typical job specification for a CMS, where the user specifies both the job and its resource requirements in one submission script, and the CMS provides the resource broker and scheduler as well as a job manager.

Thus, problems arise when jobs are submitted to the GRAM using standard RSL that has no resource requirement specifications. The GRAM passes this information on to the local scheduler using the scheduler-specific plugin, but this does not include resource information. Even if information on resource requirements is added to the RSL job submission script, it is thrown away by the GRAM and not passed to the CMS. This is not a problem if the cluster has homogeneous nodes, but for a heterogeneous cluster resource requirement information is useful (and in many cases, necessary) for the CMS to be able to effectively schedule the job.

Another potential problem with the GRAM is inefficiency. Submitting and executing jobs on a remote system will inevitably introduce some inefficiency. However, in the common case of parametric modelling, where many independent jobs are submitted at once, this overhead could become significant if the time taken for each job is not large. Most CMSs support parametric modelling by providing a mechanism for queueing multiple jobs with multiple input parameters and output data. However the GRAM supports only individual jobs, so for example when using Nimrod/G or Condor/G to queue multiple runs of the same executable with different input parameters, the job specification is translated into multiple separate GRAM jobs.

The inefficiency in this method mainly comes from two places:

- Files need to be transferred before/after each job execution. The same files used in each execution, such as the executable, cannot be shared by multiple jobs, and must be staged in repeatedly.
- Mutual authentication must be performed for each submission and file staging.

The use of a simple, standard format for specifying jobs to the GRAM is necessary for interoperability on the grid. However it comes at the cost of potential loss of flexibility and efficiency. Ideally the GRAM would also offer a mechanism for handling more advanced job information when the client application and the local scheduler support it.

For example, CMS users are familiar with the syntax of the CMS job submission script and have applications already set up to use them. It would provide them with an easier transition to grid computing if they could submit a job to the grid using a mechanism similar to the CMS. However it is by no means an easy matter to integrate CMS job submission and monitoring with Globus. Currently there are just a few such systems, such as Condor/G (Condor Project 2002) and Nimrod/G (Abramson et al. 2000), but they are typically not as easy to use as the standard CMS and have various limitations or efficiency problems. Some of these approaches (e.g. the Condor glidein mechanism) effectively bypass the GRAM. Condor is a special case that has built-in support in the GRAM to handle `jobType=condor`, as well as an alternate job manager, `jobmanager-condor`. Ideally the GRAM would support a general approach that worked for any CMS. Our preliminary experiments investigating such a system are described in section 5.

3 An Extensible Architecture for Arbitrary Execution Environments

Here we propose some mechanisms for addressing the problems discussed in the previous section, by defining some extensions to the standard RSL supported by the GRAM, as well as some modifications to the design of the jobmanager that would enable it to support arbitrary execution environments.

1. The standard RSL fields for job submission to the GRAM should be extended in order to support file staging of specified files. The new fields might be called `stagein` and `stageout`. They are GASS enabled and are used to indicate what additional files need to be transferred before and after the job execution.

2. The GRAM should pass all RSL fields in the job specification to the scheduler-specific plugins, not just the standard fields. This would allow for jobs to be specified in a more detailed way, for example resource requests, that might be usefully utilized by certain schedulers. Some schedulers (e.g. fork) may just ignore this additional information, but at least it is available to schedulers that can use it.

3. The `jobType` RSL field could be utilized to define a special execution environment, such as MPI, Java, Perl, etc. This would avoid the ugly workaround of having to use the `executable` field to define the program to instantiate the execution environment, and specify the actual program executable in the `arguments` field. Known values of the `jobType` field that are supported by the standard GRAM (i.e. a

module exists to handle jobs of that type) could be defined in the GRAM API. There should also be a standard API for the GRAM to interface to modules supporting arbitrary job types, similar to the scheduler-specific plugin API. It would also be possible to extend this to support arbitrary job types, as long as there was a well-defined mechanism for the GRAM to dynamically find and instantiate the module supporting a specified job type. This could easily be done by a Java GRAM, which could define a standard jobmanager base class that could be extended by classes specific to particular job types, which would be instantiated as required.

4. A new RSL field (which could be called `parameters`) should be defined to specify the parameters to be passed to the execution environment, e.g. the JVM memory size and classpath in Java.

5. It may be useful to define an additional set of standard RSL fields for each job type (see for example the proposed use of a `codebases` field for Java jobs in section 4). Only the module that supports a specific job type would need to understand these fields.

6. The `jobType` field could also be used to specify that the submitted executable is actually job submission script for a particular CMS, e.g. Condor, PBS, Sun Grid Engine, etc. However it would be more appropriate to define a new field (e.g. `scheduler` or `schedType`) to avoid overloading the meaning of `jobType` (see section 4 for further discussion on job managers or schedulers versus execution environments). Currently the `jobType` field in RSL can refer to an execution environment (mpi) or a scheduler (condor), which is somewhat confusing.

An alternative approach to specifying execution environments using an RSL field such as `jobType`, and providing modules callable by the GRAM to support these execution environments, would be to provide a special-purpose job manager, e.g. `jobmanager-java` to support Java programs, `jobmanager-python` for Python programs, `jobmanager-mpi` for MPI programs, etc. We believe this is not an optimal solution for several reasons. Just in a software engineering context, some of the functionality would be common to all the jobmanagers, so this is needless repetition of code, with its associated software maintenance issues. Also, it again blurs the distinction between job managers (or schedulers) and execution environments, and may lead to problems when submitting e.g. an MPI job to a PBS scheduler or a Java job to a Condor scheduler.

Another issue in implementing support for applications requiring special execution environments is how a grid application or a resource broker would discover which machines in the grid support the required environment. For example, an application that used Java would need to know which grid nodes supported Java, as well as additional information such as what version, and how to invoke the Java Virtual Machine (i.e. the full pathname of the `java` executable). Globus automatically loads information on all available jobmanagers into the GIS. It would be possible to also set up a mechanism whereby information on the software for supported execution environments (i.e. job types such as MPI, Java, etc) could also be uploaded automatically into the GIS.

4 Support for Java Applications

First, we set the context in which our work on grid support for Java applications has been developed. It emerges from the PAGIS project (Webb, Wendelborn & Maciunas 1999), an architecture for grid programming. PAGIS provides: a high-level computational model, to facilitate reasoning about behaviour at an abstract level; and a simple, intuitive, compositional API, for both direct programmiing of applications and building them with a visual program development environment.

PAGIS uses a reflective metalevel programming architecture to provide an interface for observing and influencing application behaviour; this *behaviour interface* complements, but is orthogonal to, the application programmer interface (API) through which the base-level program is composed. Various aspects of implementation, such as placement of tasks on the grid, performance, and especially adaptation, are effected through the behaviour interface.

An important aspect of PAGIS is making decisions about where on the grid to place work, through processes of resource discovery, initial allocation, and reallocation through adaptation. PAGIS uses Globus tools to facilitate such placement. PAGIS over Globus was demonstrated at CCGrid2002 in Berlin.

A PAGIS-built application comprises cooperating tasks, allocated as threads running on many distributed virtual machines. We can regard these threads and JVMs as constituting a peer group. Current Java and Globus infrastructure makes this difficult. We proceed to describe the problems, and how they can be mitigated by our proposed generalized jobmanager, and a mechanism termed Coglets (Webb & Wendelborn 2002).

The PAGIS allocator attempts to establish a suitable peer group by locating suitable Java-enabled resources, and starting Java programs. It delegates as much of this task as possible to Globus (we use Globus primarily for deployment to specific resources of executables and other files, aspects such as resource brokers for location of "best" resources are implemented separately). Hence it is important that Globus understand Java resources. Java programs are dynamic: bytecode can be loaded on demand, peers may share data and code, and there is the potential to load malicious code.

There are problems with existing techniques using the CoG, whereby we copy all bytecode to Globus resource, and start the virtual machine. A typical example of RSL was shown in section 2.3. This RSL specifies the execution of a method `Main` in a JAR archive `classes.jar`, but the manner of specification is not intuitive to either Java or grid programmers – the Java idiom is hidden by convoluted expression. In addition, we cannot express all that is important for Java-based grid resources: is a JVM installed? what version is it? precisely how is the JVM program invoked? Further, it assumes that all code is already present at runtime, counter to the Java expectation of code downloaded on demand.

Thus, we have a class of applications that requires downloading and executing arbitrary code. Further, we must consider security issues. It is clearly unsafe to not use a security manager, yet the usual Applet/RMI security managers are too restrictive in that they allow only a single code source, while our peer-to-peer applications have potentially multiple code sources.

The Coglet is our proposed infrastructure for peer-to-peer using the Java CoG. It comprises an intuitive API for submitting Java to a Globus resource; support for multiple codebases; and classloading from mutually authenticated hosts. Implementation as an integral part of the PAGIS system is underway.

Figure 1 shows the Coglet infrastructure. We see a peer group of several computation resources, each able to load code dynamically from any of several codebases, and able to securely share code, as well as data.

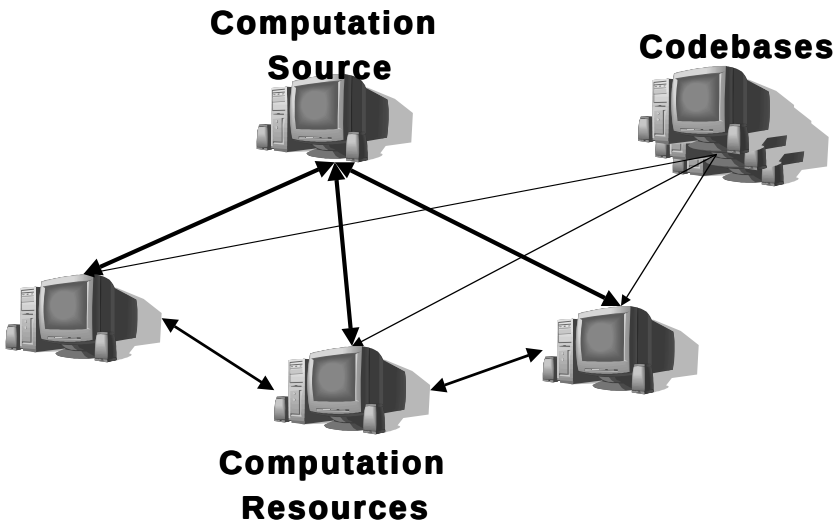

Figure 1: Coglet Infrastructure

More specifically, we proposed (Webb & Wendelborn 2002) an extended RSL specification for the abovementioned request, in the form:

```
&(executable=Main)
 (codebases=https://codebase:666/classes.jar)
 (parameters=-Dname=value) (arguments=arg1)
```

We have introduced two new attributes (`codebases` and `parameters`) and modified the interpretation of the other two. The `executable` attribute now specifies a class containing a `main()` method, denoting the name of a Java program. Optionally, this attribute could specify a JAR file with a manifest entry denoting a class containing a `main()` method. The `codebases` attribute specifies the classpath as a comma-delimited list of URLs denoting the search path for classes. Each URL specifies a directory or JAR file. The `parameters` attribute enables the programmer to specify additional virtual machine parameters, including system properties and virtual machine configuration. Finally, the `arguments` attribute specifies input arguments for the program.

We feel the new request is more intuitive to Java developers, and better fits the definition of the request attributes. However, addition and redefinition of the request requires a specialized jobmanager capable of correctly interpreting the attributes.

Previously (Webb & Wendelborn 2002) we proposed a `jobmanager-java` for Java programs, that interprets these request arguments and forks a virtual machine program with the correct arguments. A dedicated jobmanager for Java programs offers a number of benefits. The jobmanager-java implicitly identifies a Java virtual machine, independent of its implementation or file system location. The GRAM automatically publishes information on job managers to the grid information services, so a jobmanager-java would identify Java-enabled grid resources to users without the guess work. But importantly, the virtual machine program can be configured to apply security policy to code mobility.

We now show how to accommodate these ideas in the generalized framework introduced in the previous section. We can take the RSL suggested above, and add to it the clause `jobType=java`. The generalized jobmanager will then initiate execution of the requested Java code on a JVM resource, as described below. A peer group of Java resources can then be established as required, by an application such as PAGIS.

Now consider the relationship between our new Java jobmanager and the jobmanager modules of the generalized framework, and how Java code is executed. First, we distinguish between the jobmanager itself, and the *execution environment* that it sets up. For example, we might use a Condor jobmanager module to interpret RSL (such as above) that specifies running a Java executable on a new JVM – the point of the generalized framework is that it decouples the two aspects, and allows any of the jobmanagers (Condor, PBS, fork, and so on) to establish a required execution environment (Java, MPI, C executables interacting via sockets, and so on). Hence, the extended RSL suggested above is first translated into a form recognized by an appropriate jobmanager module, which then arranges deployment of the executables and associated files. Specifically, if only `jobmanager-java` is specified, the GRAM will assume a job type of Java and use the default jobmanager; this can be overridden by specifying, for example, `jobType=java, jobmanager=condor`, in which case Condor will be used to establish the required execution environment.

It is apparent that we need a precise delineation of the functions of a *job manager*, and the *execution environment* that it establishes. We can regard a *job manager* as a "deployment agent" capable of interpreting relevant RSL and setting up the specified components of the job. An *execution environment* is the execution regime under which a job op-

erates once it has commenced execution. Essentially, the job manager establishes the initial configuration of the execution environment, and thus needs to be aware of some aspects of the execution environment that it is establishing. We use the `jobtype` to inform the job manager of the underlying execution environment; the job manager has encoded within it the necessary information to establish the execution environment specified. So far, we regard this information as being hard-coded into the job manager framework. Our next step will be make our job manager *extensible*, by designing a description format for job-type-specific information, and making it possible to add a new job type and execution environment by providing such descriptions.

Further, an application, or other user of the grid, is most concerned with finding resources associated with the execution environment of the job submitted; hence, as we point out above, pertinent attributes of execution environments supported by resources should be recorded in a GIS.

An important aspect of Coglets is that it recognizes the importance of advertising, in Java terminology, the availability of Java-based resources: in other words, GIS entries with Java-specific information so that applications can request resources based on such attributes as Java version. In our generalized framework, we must extend this notion to enable insertion in a GIS of information pertaining to all supported job types, so that a resource broker can respond to requests for resources with attributes applicable to a particular job type.

More specifically, the idea behind Coglets is to provide a layer for programmers to more easily program Java for the grid, and ultimately an architecture that can be applied to other net-oriented languages.

This layer requires simple jobmanager functionality (provided by the mechanisms above) – the ability to copy a JAR file to a remote location and start a JVM with specified parameters and classpath. The Coglet layer provides secure classloading and socket communication. To this end, we also propose an extended security model for Java, whereby we can exploit Globus security to make secure Java-based grid programs. Under our proposal, peers can mutually authenticate using GSI, establishing a trust relationship between peers. A "ring of trust" is thus established among the members of a peer group. Further, codebases can be downloaded over GASS, which provides data integrity by default, and confidentiality if necessary. It is also possible to have flexible user-defined security policies, and to manipulate security policy in various ways, such as allowing code from certain otherwise untrusted hosts.

The Coglet infrastructure solves a problem encountered currently by peer-to-peer Java applications such as PAGIS – existing mechanisms for dynamically downloading code are inadequate for the needs of such applications. The RMI classloader permits it to some extent, but only for RMI invocations, which is inadequate because not all applications wish to use RMI. Applet mechanisms assume one codebase, whereas we need several. Coglets provide the required extensions. The Coglet classloader will be loaded as the system classloader, which is initiated for all classloading, not just for RMI. If any class is not located locally, the Coglet classloader will search remote codebases, download the class and reconstruct the class. Such a classloader is under development, based on Java's URLClassLoader. Downloading will be done using GASS services over GSI/SSL sockets, using the Globus certificate proxy to connect to GASS servers that each serve a codebase.

5 Support for Cluster Management Systems

To explore the suggested proposals for extending the GRAM's capabilities, particularly for file staging and for job submissions based on an arbitrary cluster management system (as outlined in section 2.4), we have implemented modified versions of the Globus `jobmanager` and the `globusrun` utility for submitting a job to the GRAM (Lu 2002). The programs were developed using the Java CoG Kit (von Laszewski et al. 2001), which provides access to Globus services through a Java framework.

5.1 Job Submission

On the client side, we developed a grid-enabled CMS job submission program called `cmsrun`, which is based on the standard `globusrun` utility, but can use the proposed extensions to the GRAM protocol. `cmsrun` is different to `globusrun` in that it aims to provide a higher-level service and a simpler interface, through which users can submit a CMS script to remote Globus resources directly. `cmsrun` uses a CMS interpreter to manipulate the CMS job script. A Java abstract class handles most of the basic features that are common to all cluster management systems, and this class can be extended by an interpreter for any particular CMS. We developed a prototype system that works for Condor.

To use `cmsrun`, users only need to specify the target GRAM server, the CMS script, and the name of CMS using the `jobType` option. `cmsrun` can choose the optimal job manager scheduler based on the given job type (i.e. CMS), manipulate the CMS script, and produce an RSL job request for it. This is explained by the following example Condor job submission script `hw.sub`:

```
initialdir   = /users/cs/condor/test
executable   = hw
input      = hw.in.\$(Process)
output      = hw.out.\$(Process)
requirements = arch == "ALPHA" && \
               opsys == "OSF1"
rank       = memory
queue 2
```

This is submitted using `cmsrun`:

```
% cmsrun <server> -jobtype condor hw.sub
```

First, `cmsrun` figures out if a jobmanager and scheduler system with the type specified in the request exists on the specified GRAM server (here `<server>` would be replaced with the actual name of a machine on the grid). In this example, `-jobtype condor` expects `jobmanager-condor`. If the desired service (`jobmanager-condor` in this example) is not available, the default jobmanager (fork) is used. In that case `cmsrun` translates the Condor file into RSL expressions, and submits them one by one, just as Condor/G handles globus universe jobs. If the desired service is available, `cmsrun` manipulates the Condor script and produces a RSL expression like:

```
&(environment = (GLOBUSRUN_GASS_URL
 http://mybox.cs.adelaide.edu.au:1799))
 (* GASS is started and its URL used by GRAM server *)
 (executable = hw.sub)
 (jobtype = condor)
 (stagein = /tmp/hw.sub hw hw.in.0 hw.in.1)
 (stageout = hw.out.0 hw.out.1)
 (count = 1)
```

Notice that files to stage in must include the Condor job script, executable, and all the input files. Moreover, instead of the original Condor job script `hw.sub`, a newly created version (written to `/tmp`) is used instead, which encapsulates some local information such as the local working directory (`initialdir`). Finally, this RSL request is submitted to `jobmanager-condor` on the specified execution host.

Both the globus universe and glidein of Condor/G can submit jobs to any GRAM server, no matter how it was implemented, however their scalability is poor. glidein can only add one Globus node at a time. To use all the nodes of a whole remote cluster, each of them must be Globus enabled and added by glidein individually. The scalability of Condor/G globus universe is mostly dependent on the job manager implementation and the composition of the remote cluster. `cmsrun` can explore all resources, even heterogeneous, within the remote cluster as if it was accessed locally.

If a resource broker is used to automatically provide the name of a valid execution host to be used with `cmsrun`, it would be possible to specify that hosts running the desired CMS should be considered in preference to hosts that do not.

5.2 Job Execution

To implement the proposed extension to the GRAM server, two kinds of programs must be modified. One is the jobmanager program, which forms the main body of the GRAM; the other is the scheduler-specific plugins that are implemented with the local scheduler system, e.g. `globus-script-condor-submit`. The modified jobmanager performs a job request in three stages. The procedure is illustrated using the `cmsrun` example.

1. When the jobmanager receives a request and recognizes the RSL expression contains file staging attributes, `stagein` or `stageout`, it will create a working directory and use GASS to transfer input files to the newly created directory. It then passes the RSL string to the plugin implemented with the local scheduler system, which is `globus-script-condor-submit` in this example. The jobmanager then monitors the execution in the usual way.

2. The internal submission API checks the `jobtype` attribute of the RSL request. If that is identical with its type, the executable is submitted to the scheduler system directly. In this example, the Condor job submission script `hw.sub` is submitted using `condor_submit`. Otherwise, the submission plugin just performs its standard function.

3. When the job finishes successfully, the job manager transfers output files specified in `stageout` back to the user and cleans up the working directory.

Note that our extended GRAM supports staging of specified files, and the submission of jobs specified as CMS scripts if the remote scheduler supports this.

5.3 Performance

The performance problems discussed in section 2.4 can mostly be solved by the proposed extensions to the GRAM. Most of the differences are fundamentally related to the local-driven dispatch of the extended GRAM and the remote-driven dispatch of Condor/G. Local-driven dispatch means jobs dispatching and executing within the local cluster, while in remote-driven dispatch, jobs are dispatched by the local system but executed on a remote cluster.

Both the extended GRAM and the globus universe of Condor/G transfer input files from the local system to the remote system before the job executes, and output files from the remote system to the local system after the job executes. However, our extended GRAM transfers multiple files at one time, which reduces the inefficiency introduced by multiple stages and mutual authentication in the globus universe of Condor/G, and avoids staging the same file repeatedly for each execution. The glidein of Condor/G performs file I/O by remote system call, a facility of the Condor standard universe, in which all file operations are transmitted back to the submit host, which can be a performance bottleneck.

6 Conclusions and Further Work

The GRAM is one of the most important parts of the Globus Grid Toolkit. In this paper, we have outlined our concerns that the GRAM lacks certain functionality that would make it easier to develop grid applications, particularly those requiring special execution environments, such as Java applications.

Some issues, such as supporting staging of specified files, can be easily addressed by simple extensions to the existing GRAM definition and implementation that would be useful for a variety of grid applications.

Providing satisfactory support for special execution environments such as Java requires significantly more work. The GRAM's current mechanisms for dealing with these environments are limited and rather ad hoc. A simple, well-defined, extensible mechanism is needed. We have presented a potential approach to this problem that requires some additions to the standard GRAM job definition and the specification and implementation of the jobmanager. There also needs to be some standard specification of precisely what information about the execution environment (e.g. the Java Virtual Machine) should be added to the GIS. We have shown how to extend current mechanisms by directly encoding, into the GRAM, support for specific individual environments. Developing a software architecture that is dynamically extensible to support an arbitrary execution environment, rather than hardwired to support a known set of environments, is a more complex problem. However this is reasonably straightforward in Java, for example by dynamically loading a class with a name specified by the requested jobType.

We have also examined the issue of enabling grid applications to dynamically download and execute code on demand. This is a particularly useful feature of Java for developing distributed applications. We have discussed our ideas for developing a framework we call coglets that would provide this capability in a Globus grid environment.

We are currently developing proof-of-concept implementations of our proposed improvements to the GRAM. This work is being carried out predominantly using the Java CoG and Java GRAM. We have a prototype client and server that supports file staging of specified files, as well as some of our ideas on support for arbitrary cluster management systems. We have done some preliminary work on exploring improved support for Java applications, however this is now being rewritten to more fully implement the ideas discussed in this paper. We are also working on the software design for a prototype of the Coglets framework, which will extend the Java grid framework to

allow for dynamic downloading of distributed codebases.

References

Abramson, D., Giddy, J. & Kotler, L. (2000), High-performance parametric modelling with Nimrod/G: Killer application for the Global Grid?, *in* 'Proc. of IPDPS 2000', Cancun, Mexico.

Condor Project (2002), 'Condor - high throughput computing', http://www.cs.wisc.edu/condor/condorg/.

Foster, I. & Kesselman, C. (1997), 'Globus: A metacomputing infrastructure toolkit', *The International Journal of Supercomputer Applications and High Performance Computing* **11**(2), 115–128.

Foster, I. & Kesselman, C., eds (1999), *The Grid: Blueprint for a Future Computing Infrastructure*, Morgan-Kaufmann.

Lu, L. (2002), Resource management for a campus computational grid, Technical Report DHPC-112, Distributed and High-Performance Computing Group, University of Adelaide. http://www.dhpc.adelaide.edu.au/reports/112/abs-112.html.

von Laszewski, G., Foster, I., Gawor, J. & Lane, P. (2001), 'A Java Commodity Grid Kit', *Concurrency and Computation: Practice and Experience* **13**(8–9), 643–662.

Webb, D. & Wendelborn, A. (2002), Java Coglets, *in* 'Proc. of Computing on Large Scale Distributed Systems (GP2PC) Workshop, Second IEEE/ACM International Symposium on Cluster Computing and the Grid (CCGrid 2002)', Berlin.

Webb, D., Wendelborn, A. & Maciunas, K. (1999), Process Networks as a High-Level Notation for Metacomputing, *in* 'Proc. of the Int. Parallel Programming Symposium (IPPS'99), workshop on Java for Distributed Computing, Puerto Rico'.