An Execution Service for Grid Computing

NAS Technical Report NAS-04-004

April 2004

Warren Smith
Computer Sciences Corporation
NASA Advanced Supercomputing Division
NASA Ames Research Center
wwsmith@nas.nasa.gov

Chaumin Hu
Advanced Management Technology Inc.
NASA Advanced Supercomputing Division
NASA Ames Research Center
chaumin@nas.nasa.gov

Abstract

This paper describes the design and implementation of the IPG Execution Service that reliably executes complex jobs on a computational grid. Our Execution Service is part of the IPG service architecture whose goal is to support location-independent computing. In such an environment, once a user ports an application to one or more hardware/software platforms, the user can describe this environment to the grid, the grid can locate instances of this platform, configure the platform as required for the application, and then execute the application. Our Execution Service runs jobs that set up such environments for applications and executes them. These jobs consist of a set of tasks for executing applications and managing data. The tasks have user-defined starting conditions that allow users to specify complex dependencies including tasks to execute when tasks fail, a frequent occurrence in a large distributed system, or are cancelled. The execution task provided by our service also configures the application environment exactly as specified by the user and captures the exit code of the application, features that many grid execution services do not support due to difficulties interfacing to local scheduling systems.

1. Introduction

The NASA Information Power Grid (IPG) project [2, 13] is one of the original grid computing projects and our goal has been to integrate, develop, and deploy a set of grid services to enable scientific discovery. The scientists we support perform tasks such as designing and analyzing aerospace vehicles, investigating the Earth's climate, and archiving and analyzing astronomical data. We have based our grid on the Globus toolkit [11] and we are currently in the process of migrating from version 2 of Globus (GT2) to version 3 of Globus (GT3). We have also deployed services such as the Storage Resource Broker [4] and Condor [5].

While we have found existing grid services to be usable, they do not always satisfy all of our needs. In particular, we have found that the collection of available grid services and software do not add up to a usable grid. There are many reasons for this, but a few examples are that users still need to know details about the resources they want to use so that they can configure their applications to use the resources and users must handle even simple failures rather than the grid handling them.

For the past two years, the NASA Information Power Grid (IPG) project has been developing higher-level grid services that attempt to create a grid to address these problems. The services we are developing include resource brokering, automatic software dependency analysis and installation, configuring execution environments, and policy-based access control. In addition, we have developed the service we describe in this paper: An Execution Service to reliably execute complex jobs in a grid environment.

The jobs sent to our Execution Service consist of a set of tasks for executing applications and managing data. A job can consist of only a few, or a large number of tasks. Our service executes the tasks in a job based on user-defined starting conditions for each task where the starting conditions are based on the states of other tasks. This formulation allows users to describe jobs that have tasks that execute in parallel and also tasks to execute when other tasks fail, a frequent occurrence in a large distributed system like a computational grid, or when the user cancels tasks. Another important feature of our Execution Service is that when it executes an application, the application is executed in the environment exactly as specified by the user and the exit code of the application is captured. This does not occur with many grid execution services because of difficulties interfacing to local scheduling systems.

This paper begins in the next section with a brief overview of the IPG service architecture and a description of how our Execution Service fits within this architecture. Section 3 provides an overview of the functionality of our

This work was supported by the NASA Computing, Information, and Communications Technology (CICT) program and performed under Task Order A61812D (ITOP Contract DTTS59-99-D-00437/TO #A61812D) awarded to Advanced Management Technology Incorporated.

Execution Service. Section 4 provides more information on the task-based job model our service supports. Section 5 describes how we are implementing our service as an OGSI service using the Globus toolkit. Section 6 presents related work and we provide our conclusions and future work in Section 7.

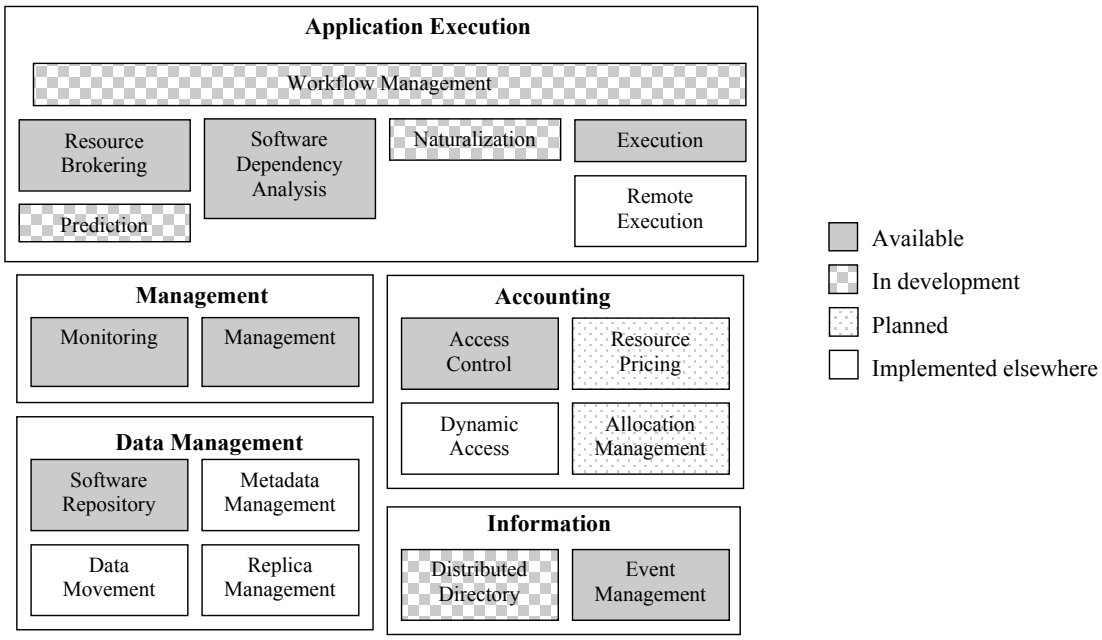


Figure 1. Architecture of the services we are creating and some of the services they interact with. Components higher in the figure tend to use components lower in the figure.

2. IPG Service Architecture

Our experience with Grid Computing has been that while there is a large amount of software available from various sources, this software does not add up to a very usable system once it is deployed. Functionality is missing from the software, the software is not as reliable as we would like, and resource differences are not hidden from our users so they end up needing to know a large amount of information about resources and their peculiarities. Our goal in the IPG is to provide a grid environment that addresses these problems and provides value to our users. To accomplish this, we are focusing on making Grid computing *location-independent*. What we mean by this is that once a user has an application that can execute on a certain hardware/software platform or platforms, the user can describe this environment to the grid, the grid can locate instances of this platform that can be used for the application, the grid can configure the platform as required for the application, and the grid can then execute the application.

Our approach to providing this location-independent environment is to build our own set of services and to use grid services implemented elsewhere. We more exactly describe our problem as providing support for location-independent execution of workflows. Figure 1 shows our architecture and provides an overview of the current status of our services. A workflow consists of set of tasks and the dependencies between these tasks where the dependencies consist of both control and data dependencies. Tasks consist of simple tasks such as those for application execution and file management and composite tasks that contain other tasks. A workflow is sent to a *Workflow Manager* to execute. The Workflow Manager decides which portion of the workflow to execute and asks the *Resource Broker* for resource suggestions for each task.

The Resource Broker makes suggestions using user-specified requirements such as resource type and user-specified preferences such as quick completion. The requirements sent to the broker describe the hardware/software platforms that are suitable for executing a task. To make selections, the Broker consults many other services. The *Distributed Directory* Service is used to search for resources with specific characteristics. The *Resource Pricing* Service is contacted to determine the cost of using these resources. The *Allocation Management* Service is used to determine if the user has an allocation that can be charged to when executing on specific resources. The *Access Control* Service is accessed to determine which resources the user can access. The *Metadata Management* Service is used to find virtual files that have the data the user requires. The *Replica Management* service is accessed to determine the physical locations of the user's data. The *Software Dependency Analysis* Service is consulted to determine what software needs to be present on a system for an application to execute. The *Software Catalog* is used to locate where needed software is already installed or can be obtained. The *Prediction* Service provides predictions of application completion times and file transfer times.

Once resources have been selected, the *Naturalization* Service is used to make each task in a workflow

compatible with the computer system(s) it will execute on by configuring environment variables, directories, and specifying any supporting software that needs to be copied to the system. The purpose of the *Execution* Service, described in this paper, is to reliably execute a task graph. A task graph is the resulting set of tasks after a workflow (or portion of a workflow) has had computer systems selected for it and has been naturalized to those systems. The Execution Service uses a *Remote Execution* Service, such as the one provided by the Globus toolkit, to execute applications on remote resources. During this execution, the *Dynamic Access* Service is used to map each grid user to a local account without the user having a pre-existing account. *Event Management* services are used by the Execution service to notify clients of the status of the execution of a task graph and are used by the *Monitoring* Service [17, 18] to notify clients of the status of resources and services. Finally the *Management* Service [17] is not visible to the general user but it received information about a grid from Monitoring services, notices when problems occur, and responds to problems in an appropriate way.

3. Overview

After several years of experience using grids, we decided that existing grid services to execute jobs did not satisfy all of our requirements for job model, job tracking, ease of maintenance, and other features. We therefore began developing an Execution Service that would satisfy our requirements and those of our users. The version presented here is the second major version of our service and it provides much of the functionality that our users have requested after using the first version of the service for almost a year.

Our Execution Service allows users to submit, monitor, and cancel complex jobs. Each job consists of a set of tasks that perform actions such as executing applications and managing data. Each task is executed based on a starting condition that is an expression on the states of other tasks. This formulation allows tasks to be executed in parallel and also allows a user to specify tasks to execute when other tasks fail or are cancelled. Our support for such complex jobs has evolved out of our previous version of the Execution Service that supported a job model of pre-stage files, execute a single application, and post stage files. Our users asked for additional functionality such as creating directories and deleting files, executing multiple applications in one job, and specifying what tasks to execute when tasks fail or are cancelled. Further information about our job model is presented in Section 4.

Our Execution Service attempts to execute tasks in a reliable manner. In a grid, resources such as networks, computer systems, and storage systems are constantly unavailable for planned maintenance and unplanned failures. Further, even when the resources are available, the software and services located on those resources may be unavailable or not operating correctly. There are ways to mitigate this inherent unreliability by techniques such as pre-planning outages and monitoring the status of a grid [7, 16] so that failures can be quickly repaired, but this will not eliminate the problem. To help our users deal with failures, our Execution Service detects when tasks fail and retries them when appropriate. To determine how to handle a failure, information about the cause of the failure is needed.

After a job has been submitted to our Execution Service, users can monitor it in several ways. While the job is executing, users can either be notified when the state of the tasks in a job change or they can query to obtain a history of state changes for each task in a job. Further, many applications indicate whether they executed successfully or not using the exit code of the application. This is important information that our service captures, provides to the user and uses to determine if the execution of an application succeeded or failed.

The notification of task state changes is accomplished by our Execution Service supporting the Event Producer interface of our event management framework and the client of our service supporting the Event Consumer interface of our event management framework. This allows the client to subscribe for events about task state from the service and the service to notify the client when the tasks change state. Another way that users can monitor their jobs is that even after a job is finished, users can query the Execution Service to obtain all of the information relating to the job. This information is stored for a user-specified amount of time with a default of several days. The ability to obtain information about a job that has already completed is very useful because it allows users to easily determine if a job that ran while the user was not watching it executed correctly. Without this historical record, a user has to examine the output of their application executions to determine if they executed correctly. If a failure occurred, a user has to use their application output to try to determine which application executions or file management operations failed.

We have implemented our Execution Service as an OGSI service [19] using version 3 of the Globus Toolkit [1]. Our service operates in a client-server manner, with the clients installed on our user-accessible systems and our service installed on a computer system dedicated to hosting grid services. We currently have version 2 of the Globus toolkit deployed on the IPG so the Execution Service executes tasks using the Globus Java CoG [14] to access the Globus Resource Allocation Manager (GRAM) and GridFTP services on our systems. Further information about our implementation is presented in Section 5.

4. Job Model

The goals for our job model are to support complex jobs consisting of many actions and support conditional execution of actions depending on the states of other actions. To satisfy these goals, we have defined a job model where a job is a set of *tasks*. Each task has:

- An *identifier* that is user-defined and unique among all of the identifiers of sibling tasks.
- A *starting condition* that describes when the task can be started. This condition is specified as a Boolean expression on the states of other tasks. A starting condition can be empty which indicates that the task can be started immediately.
- A *state* that is:
    - NOT_READY if the starting condition of the task has not been met
    - READY if the starting condition of the task has been met, but the task has not yet begun to execute
    - RUNNING if the task is executing
    - SUCCEEDED if the task executed successfully
    - FAILED if the task failed during execution
    - CANCELLED if the task was cancelled by the user
    - NOT_EXECUTED if the task will not be executed because it's starting condition will not be met
- The state transition diagram for a task is shown in Figure 2.

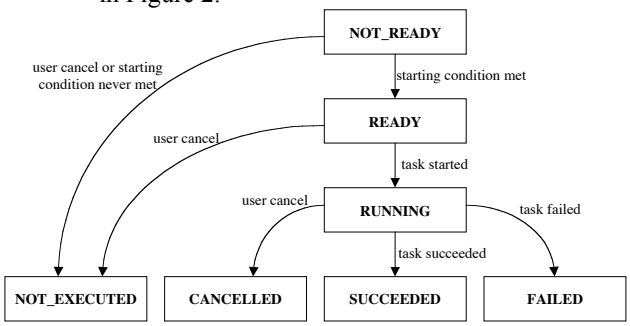


Figure 2. State diagram for a task.

We currently provide a variety of *atomic tasks* and a *composite task*. An atomic task is a relatively simple task that does not contain other tasks. We have defined atomic tasks that contain general task information (identifier, starting condition, and state) but also require additional information. We have defined the following atomic tasks:

- An ExecuteTask that executes an application on a remote computer system. A user specifies parameters such as the host to execute the application on, the application to execute, the arguments to the application, the number of CPUs, and so on. This task also has a user-specified Boolean equation on the exit code of the application so that the user can specify which exit codes indicate success and which ones indicate failure. By default, an exit code of 0 indicates success and any other exit code indicates failure. The exit code used in this equation is also provided to the user by the task.
- A MakeDirectoryTask that creates a directory on a remote computer system. This task requires a host and directory name.
- A CopyTask that copies files between remote computer systems. The user specifies source and destination hosts, directories, and file names where the file names can include wildcards. A user can also specify that a recursive copy should be performed.
- A MoveTask that moves files between remote computer systems. The user specifies source and destination hosts, directories, and file names where the file names can include wildcards. A user can also specify that a recursive move should be performed.
- A RemoveTask to remove one or more files or directories. The user specifies a host, directory, and file where the file name can include wildcards. A user can also specify that a recursive remove should be performed.

A composite task is used as a container for other tasks. The use of composite tasks allows users to group tasks that collaborate to perform a function into a single task and then consider this functionality in an abstract manner. In fact, a job submitted to the ExecutionService is simply a composite task. While the same states are used for a task whether it is atomic or composite, the current state of a composite task is determined in a specialized way. The state of a composite task is:

- NOT_READY until the starting condition of the composite task is satisfied
- READY when the starting condition for the composite task has been met but no subtasks of the composite task have started to run.
- RUNNING while any subtask of the composite task has had a state of RUNNING and any subtasks are currently READY or RUNNING

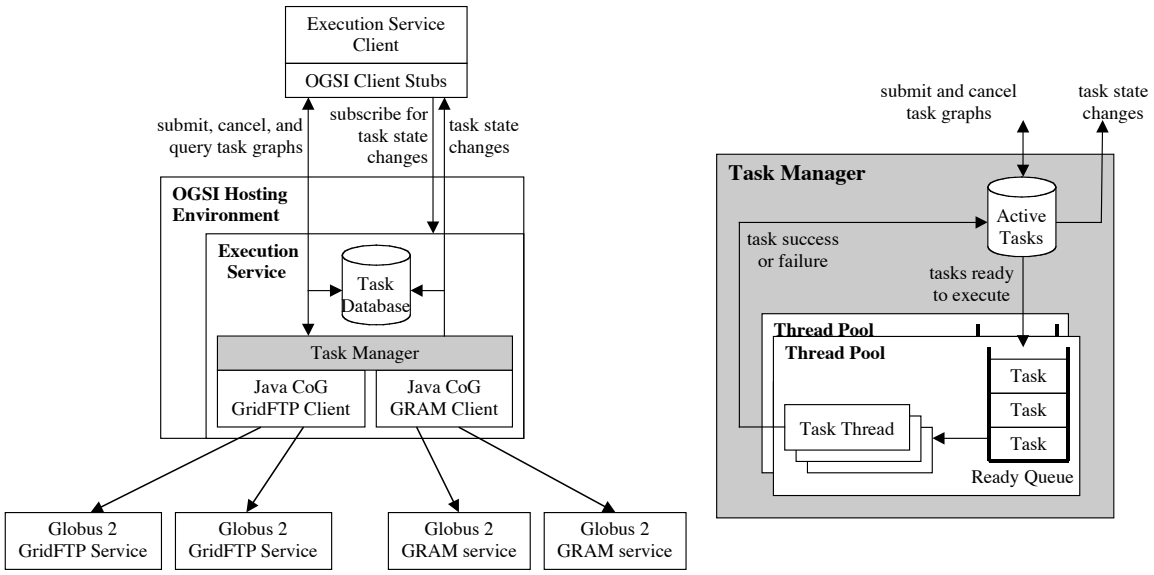


Figure 3. Overview of the implementation of our Execution Service.

- SUCCEEDED or FAILED based on a user-defined Boolean expression when no more subtasks of the composite task can run. The Boolean expression contains variables that are the states of the subtasks in the composite task. This approach provides a user with very precise control over the completion state of a task without us defining a one-size-fits-all approach.
- CANCELLED if the user cancels the composite task.
- NOT_EXECUTED if the starting condition for the composite task will not be met.

5. Implementation

We have implemented our execution service as an Open Grid Services Infrastructure (OGSI) [19] service using version 3 of the Globus Toolkit as our hosting environment. We plan to deploy only a few of these services on computer systems dedicated to hosting services and install clients on the user-accessible IPG computer systems. The purpose of this approach is to improve reliability and maintainability. Reliability is hopefully improved by having only a few services deployed on closely monitored systems. Maintainability is improved by being able to easily upgrade services deployed on a few systems rather than a service deployed on every user-accessible system. This approach was very helpful with the first version of our Execution Service because we upgraded the deployed services many times without upgrading the clients.

An overview of the implementation of our Execution Service is shown in Figure 3. The core components of our service consist of a Task Database and a Task Manager. The Task Database is used to store tasks that have been submitted for execution and is initially implemented atop a Xindice database. Users can obtain information about both active (not yet completed) and inactive (completed) jobs. Information about inactive jobs is stored for several days by default and a user can also specify the amount of time to store job information.

The Task Manager is the core of the service and handles the execution of tasks. The two main goals of the Task Manager are to execute tasks in the proper order, based on the user-specified starting conditions, and not overload local and remote resources while executing tasks. A more detailed view of the Task Manager is shown on the right side of Figure 3.

Whenever tasks are added to the pool of Active Tasks or whenever tasks finish executing, the Task Manager examines the Active Tasks and determines if any are now ready to run. These ready tasks are moved to Ready Queues in the Thread Pools to execute. By following this procedure, the Task Manager will execute tasks in the correct order.

A Thread Pool contains a set of Task Threads to execute tasks and a Ready Queue containing tasks that are ready to execute. A Task Thread removes a task from the head of the Ready Queue, executes that task, and then tries to get another task to execute from the Ready Queue.

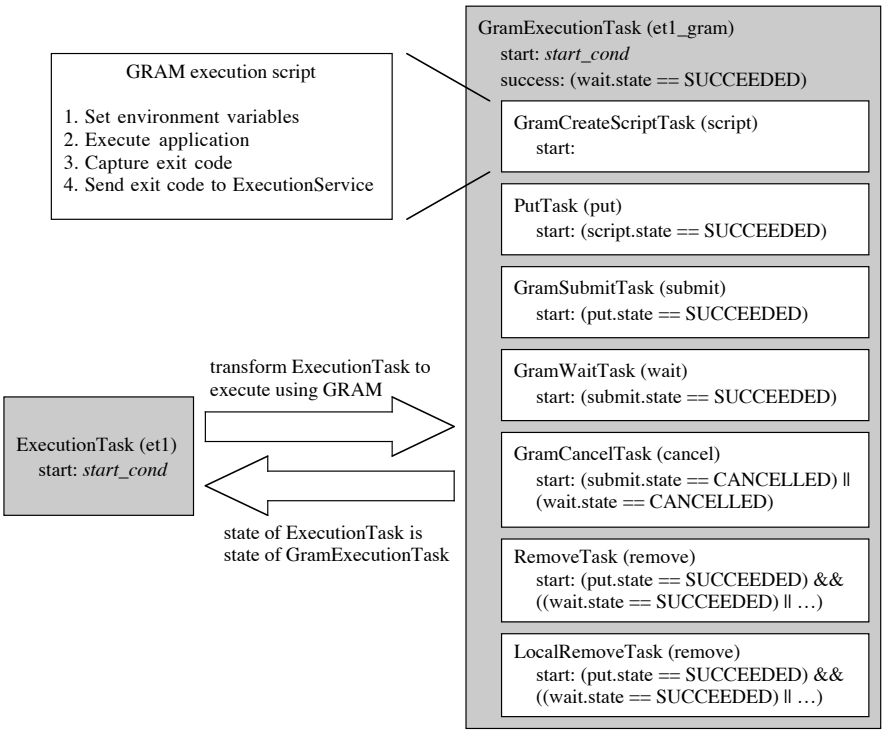


Figure 4. Implementation of an ExecutionTask using the Globus GRAM.

The Task Manager moves a task to a Thread Pool based on task type. A Thread Pool has either a fixed or an unlimited number of threads available to execute tasks. Thread Pools with fixed numbers of threads are used to execute tasks that may overload a system such as submitting applications and performing file management operations. The limited number of threads bounds the amount of concurrency and reduces the chance of overwhelming the server running the Execution Service or the resources being accessed by that service. Thread Pools with unlimited numbers of threads are used to execute tasks that will not overwhelm a resource, such as waiting for an application execution to complete.

As described next, individual tasks also use supporting software such as the Globus Java COG GRAM and GridFTP clients to perform their functions.

5.1. Executing Applications Using Globus

We use the Globus Java CoG library to implement our task that executes applications. We use the CoG GT2 clients rather than GT3 clients because we currently have GT2 services installed on the IPG. We expect it to be a simple matter to substitute calls to the Globus 3 client library calls for version 2 calls when we upgrade to GT3 services.

We use the Java CoG GRAM client to execute applications, but in a particular way. We do not use the GRAM to directly execute the application specified by the user in the ExecuteTask. Instead we execute a script that we create. We have found that the combination of the GRAM and different local schedulers results in several problems. First, environment variables are not always passed to the application as expected. If a user specifies an environment variable in the Globus Resource Specification Language (RSL), this environment variable may not be set, may be set, or may be appended to the end of the existing environment variable. In many cases, users pass execution parameters to their applications using environment variables so it is important that these variables be set correctly. Second, exit codes from applications are lost. The Globus GRAM does not attempt to return exit codes, and even if it did, local scheduling systems often do not provide exit codes that the GRAM could return to the user. In many cases, applications indicate if they have executed correctly using exit codes so it is also important that these exit codes are available to users.

Our approach to both of these problems is to create and execute a script. This script sets the environment variables exactly as specified by the user, executes the user-specified application, captures the exit code of the application, and sends this exit code to the Execution Service using our Event Management Framework. Each ExecuteTask is translated into a composite GramExecuteTask, shown in Figure 4, to accomplish this. The execution script is created by the GramCreateScriptTask and is copied to the execution host using a PutFileTask (not available to users) that uses a GridFTP put. The GramSubmitTask then submits our script using the GRAM. The GramWaitTask waits for a GRAM job to finish and the GramCancelTask is called to cancel the GRAM job if the user cancels the ExecutionTask. We use these three GramTasks because both the GramSubmitTask and GramCancelTask require authentication, which is a CPU-intensive task that can

overwhelm both the server running the ExecutionService and the computer system running the GRAM server, while waiting for a GRAM job to complete requires virtually no resources. We therefore wanted to limit the number of simultaneous GRAM submits and cancels but did not want to limit the number of GRAM jobs that the Execution Service is waiting to complete. Finally, a RemoveTask and a LocalRemoveTask (not available to users) are used to remove the execution script that we created from the remote and service hosts.

5.2. File Management Using Globus

We also use the Globus Java CoG library to execute our atomic tasks that manage files. Once again, we use the GT2 Java CoG clients because we currently have GT2 services installed on the IPG. We use the GridFTP client provided by the Java CoG to copy, move, and remove files as well as to make directories. We copy files between hosts using the 3rd party copy functionality of the Java CoG. We enhance the functionality provided by the Java CoG by maintaining the permissions of the transferred files (such as the executable bit), by supporting wildcards in file and directory names, and by providing recursive copies. We enhance the ability of the Java CoG to remove files on remote hosts by allowing users to specify wildcards in file and directory names and specifying that the remove should be performed recursively. We provide moves of files by performing a copy of the files and, if the copy succeeded, removing the files from the source host. Finally, we directly use the Java CoG to make directories on remote hosts.

6. Related Work

There are a fair number of services that support the execution of jobs on grids. The basic grid service for executing applications on remote computers is Globus GRAM [8] in both it's GT2 and GT3 incarnations. While GRAM performs it's basic function adequately, it does have some deficiencies. It does not always set the environment of the application as specified by the user, due to difficulties interfacing to the many different types of local scheduling systems. It also does not capture the exit code of applications executed through it. Finally, GRAM lacks the ability to execute complex jobs, such as the ones we support.

The Condor-G system [12] uses the GRAM service, but improves on it by enhancing it's reliability. Unfortunately, this improvement currently comes with an administrative cost of maintaining a Condor-G daemon on each host that wishes to submit Condor-G jobs. The Condor group is beginning to address this problem by providing a web service wrapper around Condor-G daemons so that remote clients can access those daemons, but a Condor-G version with this functionality has not yet been released. Our service is already implemented in a client-server manner and does not have daemons running on client hosts. Condor-G has the same goal of reliable execution as our service but it does not support jobs as complex as ours. Also, unlike our service, Condor-G does not maintain a database of jobs that have completed that users can access.

DAGMan [6] is built atop Condor-G and supports the execution of Directed Acyclic Graphs. A DAGMan job consists of a set of Condor submit scripts to execute where each script has execution order dependencies with other scripts. A script is executed when all of the scripts it depends on complete successfully. Each script may have pre- and post-execution programs to execute before and after a script is executed. If the pre-execution program fails, its script will not be executed. All post-execution programs in a DAGMan job can either be executed or not when their associated scripts fail depending on a flag set when submitting the DAGMan job. Our job model is somewhat similar to the DAGMan job model. One of the main differences is that we provide a more general approach to specifying when to start tasks with our starting condition expressions. This allows our service to handle failures in a more general way by defining complex sets of tasks to execute when tasks fail or are cancelled. Our service is also different in that it does not support the specification pre- and post-task programs to execute because they are unnecessary in our job model, we provide built-in tasks for file management, and we provide composite tasks that contain sets of tasks.

Pegasus [9] is a workflow execute tool where the user specifies the tasks to perform without specifying where to perform them, Pegasus decides where to execute the tasks and creates a DAGMan job to execute the tasks. Pegasus workflows are not as fault tolerant as ours because they do not include tasks to perform when tasks fail or our cancelled and Pegasus workflows are not as complex as ours because they do not support composite tasks that contain sets of tasks. Pegasus selects resources for the workflows submitted to it; functionality that we do not support in our Execution Service.

UNICORE [10] provides it's own services for executing jobs. These jobs are similar to ours in that they can consist of many tasks with execution order dependencies between them and the tasks can be composite tasks that contain other tasks. We provide more flexible conditional task execution than UNICORE abstract jobs, but UNICORE does allow a user to indicate if tasks should execute whether or not the tasks it depends on succeed or fail. Similar to our approach, UNICORE also maintains job information after it has completed for the convenience of users. UNICORE also provides features such as executing each job in it's own file space which is a convenient abstraction. Unfortunately, UNICORE is a vertical solution and requires adopting all or none of it.

We use GridFTP [3] to manage remote files and we add to the functionality provided by the Java CoG [14] GridFTP client by supporting wildcards and recursive operations. Our service also provides a superset of the functionality available from reliable transfer services [15].

7. Conclusions and Future Work

This paper presents our IPG Execution Service that is implemented as an OGSI service and reliably executes complex jobs on a computational grid. This service is part of our IPG service architecture whose purpose is to provide a grid environment where users can execute applications in a location-independent manner.

The jobs sent to our Execution Service consist of a set of tasks for executing applications and managing data. Our service executes each task in a job based on a user-defined starting condition that is based on the states of other tasks. An important feature of this formulation is that it allows users to describe tasks to execute when tasks fail, a common occurrence in a large distributed system like a computational grid, or when the user cancels tasks. Another important feature of our Execution Service is that when it executes an application, the application is executed in the environment exactly as specified by the user and the exit code of the application is captured, features not supported by many grid execution services.

There are several directions that we may take for future work. First, as requested by our users, we will provide C++ and Perl clients to our service. This will force us to learn a different OGSI framework, the gSOAP framework that is part of GT3, and wrap the C++ clients we create with this framework to create Perl clients. Second, we will need to support using GT3 mechanisms for executing applications and managing files once we upgrade our IPG infrastructure from GT2 to GT3. Third, there are quite a few new tasks that we could support. We could add tasks to manage files indexed by replica catalogs, to select virtual files based on metadata, to execute an application across multiple computer systems, to indicate that tasks should execute simultaneously, or to perform loops using special types of composite tasks. Fourth, we could enable group-based access to execution information. Our scientists typically work in groups so such access could be useful. Fifth, the job database contained in the Execution Service could be enhanced to provide arbitrary searches and to allow users to annotate jobs with information that they will find useful later. Finally, we could allow our users to pause submitted jobs, modify them, and then un-pause the jobs.

References

[1] "The Globus Project," http://www.globus.org

[2] "The NASA Information Power Grid," http://www.ipg.nasa.gov

[3] B. Allcock, J. Bester, J. Bresnahan, A. Chervenak, I. Foster, C. Kesselman, S. Meder, V. Nefedova, D. Quesnel, and S. Tuecke, "Data Management and Transfer in High Performance Computational Grid Environments," *Parallel Computing Journal*, vol. 28, pp. 749-771, 2002.

[4] C. Baru, R. Moore, A. Rajasekar, and M. Wan, "The SDSC Storage Resource Broker," Proceedings of the CASCON'98, Toronto, Canada, 1998.

[5] A. Bricker, M. Litzkow, and M. Livney, "Condor Technical Summary," Computer Sciences Department, University of Wisconsin - Madison 1991.

[6] Condor, "Condor Version 6.4.7 Manual," University of Wisconsin-Madison 2003.

[7] K. Czajkowski, S. Fitzgerald, I. Foster, and C. Kesselman, "Grid Information Services for Distributed Resource Sharing," Proceedings of the The 10th IEEE International Symposium on High Performance Distributed Computing, 2001.

[8] K. Czajkowski, I. Foster, N. Karonis, C. Kesselman, S. Martin, W. Smith, and S. Tuecke, "A Resource Management Architecture for Metasystems," *Lecture Notes on Computer Science*, vol. 1459, 1998.

[9] E. Deelman, J. Blythe, Y. Gil, and C. Kesselman, "Pegasus: Planning for Execution in Grids," University of Southern California, Information Sciences Institute 2002-20, November 15 2002.

[10] D. Erwin, "UNICORE Plus Final Report - Uniform Interface to Computing Resources," UNICORE Forum e.V. 2003.

[11] I. Foster and C. Kesselman, "Globus: A Metacomputing Infrastructure Toolkit," *International Journal of Supercomputing Applications*, vol. 11, pp. 115-128, 1997.

[12] J. Frey, T. Tannenbaum, I. Foster, M. Livny, and S. Tuecke, "Condor-G: A Computation Management Agent for Multi-Institutional Grids," Proceedings of the 10th International IEEE Symposium on High Performance Distributed Computing, San Francisco, CA, 2001.

[13] W. Johnston, D. Gannon, and B. Nitzberg, "Grids as Production Computing Environments: The Engineering Aspects of NASA's Information Power Grid," Proceedings of the 8th IEEE International Symposium on High Performance Distributed Computing, 1999.

[14] G. v. Laszewski, I. Foster, J. Gawor, W. Smith, and S. Tuecke, "CoG Kits: A Bridge between Commodity Distributed Computing and High-Performance Grids," Proceedings of the ACM Java Grande Conference, 2000.

[15] R. K. Madduri, C. S. Hood, and W. E. Allcock, "Reliable File Transfer in Grid Environments," Proceedings of the 27th IEEE Conference on Local Computer Networks, 2002.

[16] W. Smith, "A Framework for Control and Observation in Distributed Environments," NASA Advanced Supercomputing Division, NASA Ames Research Center, Moffett Field, CA NAS-01-006, June 2001.

[17] W. Smith, "A System for Monitoring and Management of Computational Grids," Proceedings of the International Conference on Parallel Processing, Vancouver, Canada, 2002.

[18] B. Tierney, R. Aydt, D. Gunter, W. Smith, V. Taylor, R. Wolski, and M. Swany, "A Grid Monitoring Service Architecture," Global Grid Forum Performance Working Group 2001.

[19] S. Tuecke, K. Czajkowski, I. Foster, J. Frey, S. Graham, C. Kesselman, T. Maquire, T. Sandholm, D. Snelling, and P. Vanderbilt, "Open Grid Services Infrastructure Version 1.0," The Global Grid Forum June 27 2003.