The Java CoG Kit Experiment Manager

Gregor von Laszewski,[1,2], Phillip Zimny,3, 1, Tan Trieu[4,1], David Angulo[5,1]
[1] Argonne National Laboratory, Argonne National Laboratory, 9700 S. Cass Ave., Argonne, IL 60440
[2] University of Chicago, Computation Institute, Research Institutes Building #402, 5640 South Ellis Ave., Chicago, IL 60637-1433
[3] PHILS INSTITUITION
[4] Santa Clara University, Department of Computer Engineering, 500 El Camino Real, Santa Clara, CA 95053
[5] DAVES ADDRESS DePaul University

Abstract

In this paper, we introduced a framework for experiment management that simplifies the users' interaction with grid environments by managing a large number of tasks to be conducted as part of the experiment by the individual scientist. Our framework is an extension to the Java CoG Kit. We have developed a client-server approach that allows us to utilize the grid task abstraction of the Java CoG Kit and expose it easily to the experimentalist. Similar to the defintion of standard output and standard error we have defined standard status that allows us to conduct application status notifications. We have tested our tool with a large number of long running experiments and show its usability in practical applications such as bioinformatics.

1 Introduction

Grid computing addresses the challenge of coordinating resource sharing and problem solving in dynamic, multi-institutional virtual organizations [?]. The analogy between the computational grid and the power grid highlights the emphasis on virtualization. When a user plugs an appliance into the power outlet, he/she expects the delivery of power without concern for the whereabouts of the power source. Just as the electric power grid allows pervasive access to electric power, computational grids provide pervasive access to compute-related resources and services [1]. The Grid's focus on integrating heterogeneous, distributed resources for the purpose of high performance computing differentiates it from other technologies such as cluster computing and the Web. The Grid's ability to virtualize a collection of disparate resources to solve problems promisses effortless collaboration among the scientific communities.

The construction of the Grid requires the establishment of standards for a secure and robust infrastructure. One such undertaking is the definition of the Open Grid Services Architecture (OGSA), which provides a specification for a standard service-oriented Grid framework [1]. The implementation of the services form the Grid middleware, and the Globus Toolkit is today's de facto standard Grid middleware [2]. The toolkit provides an elementary set of facilities to handle security, communication, information, resource management, and data management services [1]. However, the set of services may not be compatible with the commodity technologies that Grid application developers use. The Commodity Grid project addresses the incompatibility by creating Commodity Grid (CoG) Kits that define mappings and interfaces between Grid services and particular commodity frameworks such as Java, Perl, and Python [1].

The Java CoG Kit provides more than just a mapping between Java and the Globus Toolkit. The Java CoG Kit bridges the Java commodity framework and Grid technology. This means it not only defines a set of convenient classes that provide the Java programmer with access to basic Grid services [3], but also integrates a number of sophisticated abstractions, one of which is a workflow system [4]. Hence, it provides a significant feature enhancement to existing Grid middleware [1].

A popular use of the Grid is motivated by the field of bioinformatics, where applications such as Grid-enabled Blast [?] are used to compare base or amino acid sequences registered in a database with sequences provided by the user [?]. Blast runs can generate numerous queries that require hours or even days to complete. Managing such studies requires that scientists maintain the status and outputs of the

individual queries, distancing them from the experiment at hand and burdening them with the tedious task of checking for job status and output. In an effort to relieve the scientist from the drudgery of managing output data and provide the scientist with a tool to monitor the progress of his/her jobs, we introduce the concept of an experiment. An experiment can be defined as tasks that are executed on the Grid with their associated output stored in a user-defined location. In this paper we show that the Java CoG Kit is ideally suited to support such a high level service. Using the facilities provided by the Java CoG Kit, we create a user driven experiment management system to simplify the administration and execution of repetitive tasks that use similar parameters.

The user driven experiment management tool combines features of several tools to empower the novice Grid user. It includes features typically found in queuing systems, shells with history, and process monitoring programs such as the well known UNIX ps command. Naturally it is targeted to include specific enhancements for the Grid environment. To emphasize on the similarities let us revisit a typical use case of a user using a UNIX command shell. A user working in the UNIX command shell queries which jobs have been submitted by the history function. The status of process, a running instance of a program, can be obtained by issuing the ps command. The output provides information such as the process ID, current status, the cumulated CPU time, and executable name. Our experiment management system provides a similar interface, displaying the added experiments in a format that includes the experiment ID, current status, cumulative time the experiment has been queued, and experiment name. However, in extension to the normal command and history management tool by shells we must integrate user accessible outputs and error files on a command by command basis.

A preliminary version of the execution manager is already available for years as part of the Java CoG Kit under the name Grid Command Manager (GCM). However, we enhanced its functionality significantly.The enhancements include experiment status checkpointing, management support for a large number of experiment submissions, and the integratin of fault tolerant queues for managing experiment submissions.

The rest of the paper is structured as follows. First, we revisit our requirements that lead us to a redesign of the Grid Command Manager for experiment support. This includes the presentation of a use case for our framework. Next we describe the architecture that fulfills our requirements. We describe the implementation and present preliminary performance results. We conclude the paper with our thoughts on future work to be conducted.

2 Requirements

The experiment management system has several major requirements including automated experiment checkpointing, transparent output management, automated version control, metadata management, detailed status reporting, persistent experiment sessions, scalable experiment updating. Next we will discuss each of the requirements in more detail.

Automated Checkpointing. A basic assumption that the experiment management system makes about experiments is that they are non-interactive, long running jobs. With long running experiments, the expectation that the host requesting the remote resource maintains an uninterrupted connection with the remote resource is impractical. From this stems the requirement that checkpointing, or saving the state, of an experiment must be a transparent process so that users do not have to associate experiments with checkpoint files. Instead, once a user submits an experiment he/she must only associate the experiment with its name in order to track its status. To address the overhead of maintaining a persistent connection, the Java CoG Kit abstractions module provides a checkpointing mechanism that enables users to reconnect to a submitted job at a later time.

Transparent Output Management. To shield the user from details about the Grid, the standard error (stderr) and standard output (stdout) are automatically saved in a predetermined experiment path location to prevent the impression that the stdout and stderr have vanished because they reside on the remote execution host or because the experiment has been duplicated (see also Version Control). Such functionality provides the illusion of localized computing while using the Grid.

Version Control. Storage of output files leads to the next requirement of output version control. When an experiment is submitted more than once, the output from its previous runs needs to be stored and

accessible for comparison to its future runs. An automated version control system sequentially names versions of output. This is to take the responsibilities of re-naming, moving, and organizing different versions of output away from the scientist.

Metadata Management. A scientist often has additional information about an experiment that needs to be managed. Such information includes the authors of the experiment, the date the experiment, and other information pertinent for organizing and documenting of an experiment. Hence, an additional requirement is to provide a system to automatically maintain the metadata of each experiment. This system must allow for easy entry of an experiment's metadata as well as allow for changes to be made. It will allow the scientist to reference more than just the output to uniquely identify each experiment.

Application Status Reporting. [why we have done this] Besides retrieving stdout and stderr, we believe that users will benefit from application status reporting. Similar to stdout, we introduce a standard status (stdstatus) that can be used to report more detailed experiment states as well as application specific information.[1] When checking the status of an experiment that reached the failed state the user may wonder what triggered or caused the failure. The standard status provides this service by trapping signals that interrupted the job. The user can then query the standard status to review the events that occurred before the failure.

The use of the standard status goes beyond error reporting; it provides a simple technique for runtime application status notification. For experiments that take days to complete, knowing that the experiment is running is often inadequate. The standard status provides a mechanism for application developers to expose a more detailed record of the application's progress during execution.

Persistent Experiment Sessions. The ability to load information about previous experiments when restarting the experiment manager is an important maintenance tool. In case the experiment manager abruptly shuts down or if the user has multiple instances of the experiment manager running, persistence enables the user to maintain sessions.

[1]this is an implementation detail

Scalable Experiment Status Updating. With persistent sessions, the number of experiments within a session can grow quite large. The task of updating the status of such a large number of experiments can consume a disproportionate amount of computing resources on the client machine. The experiment management system thus needs to simulate thread execution when updating the status of each experiment, rather than creating a thread dedicated to status updating for each experiment.

3 Use Case

[focus on biology case; derive requirements] In this section we describe a scenario for the use of the cog-experiment tool.

The Basic Local Alignment Search Tool (BLAST) is one of the most popular tools for searching nucleotide and protein databases. It tests a nucleotide sequence against a database of known sequences and returns similarities. BLAST offers many different types of queries to the data base including ones such as nucleotide to nucleotide, protein to nucleotide, protein to protein, nucleotide to protein, as well as many more. Biologists use this tool to help them discover the identity of the sequence they are studying or to identify the function of the sequence they are studying by comparing it to similar sequences BLAST finds.

A biologist may find themselves in the scenario where they wish to research a specific genetic sequence by making 500 slight modifications to the sequence, run the it through BLAST and see if the modification produces any similar characteristics of other sequences. The biologist would have start by running the original sequence and then run each of the 500 modifications. At the beginning of each submission of the BLAST run, the biologist would have to name the task himself and have to then keep track of the 500 different names. Upon the completion of the BLAST run the biologist would then have to move his output files into separate directories which they would have to name and then remember.

Once the biologist has completed his 500 BLAST runs, he now has 500 different outputs to manage. The biologist most likely does not wish to devise a method of organizing all of the different output they have created. Even once organized, there is very little addition data associated with the outputs to allow the biologist to search through the output files. This research method is inefficient and timely and not the

optimum way a biologist would like to conduct their research.

The cog-experiment tool offers a way to make this process not only much simpler but also much more efficient. This tool allows the biologist to set up the submission process to repeat itself however times is need, in this case 500, while making a slight modification to the submission parameters each time. For this example, the modification to the parameters would be the file name each modified sequence was stored in. The submission process no longer requires the biologist's presence. If these submissions took long periods of time to complete, such as several days, the cog-experiment tool would also checkpoint progress on the completion of a submitted task so the researcher would not have to start the task over from the beginning.

In addition to offer a better submission procedure to the biologist, the cog-experiment tool simplifies the file management for the biologist. The biologist may pick one name and the cog-experiment tool will automatically assignment a sequential version number to each submission. Starting with one, the biologist now has an easy to understand version schema. This auto-naming feature takes away any worries about over writing data or forgetting the names used for submission.

Once each submission has been automatically named, a folder of the same name is created to store that individual submission's files in the user's experiment path location. Now the biologist has all of their submission files properly named and has the output file neatly stored in individual folders.

The biologist can use the option to enter metadata about the submissions on an individual level. Information about the specific submission such as author, time, or other notes can be saved to persistent storage. When the biologist does this, it allows them to search through all the outputs. For example, if the biologist wishes to view all the submissions from a certain day, they can simply search for that date and all of the submissions from that date are displayed on the screen.

4 Architecture

The architecture of the experiment management system integrates with the Java CoG Kit's layered approach. The experiment management system is a module that reuses the abstractions layer while exposing a command line tool. The abstractions layer provides high level abstractions that include Grid tasks, transfers, jobs, and queues that make developing Grid programs easier [4].

The experiment management system consists of two primary components, an experiment manager and a command line component. These components communicate via a socket, with the experiment manager running as a background job that services requests from the command line component to add, remove, submit, list, and retrieve the status of experiments.

Figure 1 depicts the architecture of the experiment management system. The heart of the system is the experiment manager component, which maintains experiment status with a set of four queues: pending, submitted, completed, and failed. An experiment's transition through the queueing system is illustrated through the state diagram in Figure 2. The user has control over two state transitions: adding an experiment to the pending queue, and performing a local submit to move the experiment to the submitted queue. The rest of the state transitions are handled by a background thread, which periodically updates the status of the queued experiments.

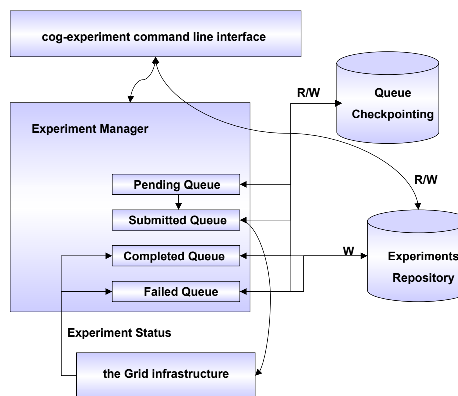

Figure 1: The architecture of the Java CoG Kit experiment management framework.

The experiment manager uses persistent storage to provide automated experiment checkpointing, transparent output management, and persistent experiment sessions. The automated experiment checkpointing and transparent output management functions rely on an experiment repository to store the checkpoint files for each submitted experiment and to save the stdout, stderr, and stdstatus resulting from

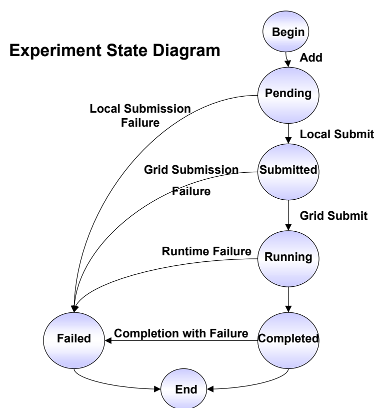

Figure 2: State diagram of an experiment as it transitions through the four queues.

an experiment. To provide persistent experiment sessions function, the experiment manager periodically checkpoints the status of the four queues to persistent storage, where the status of the four queues can be reloaded when the system is restarted.

The cog-experiment command line interface component provides access to the experiment manager functions to add, remove, submit, and retrieve the status of experiments. The command line interface also provides other functions such as metadata maintenance and version control, and thus requires access read and write access to the experiments repository.

[figures changes: class relationships - fill out page redraw arch w/ lines

5 Implementation

The implementation of the experiment management system is split into the client and server components.

5.1 client

make sure that client and server clearly separated in different subsections Experiment manager client is implemented using an argument parser to retrieve the user's desired function. Once the user has entered their command experiment manager client then parses the command into its separate components to determine which method to call. Experiment manager client's methods use instances of experiment metadata implementation and experiment output manager. These two classes both use instances of experiment data manager to help organize an experiment's metadata.

[clarity on client/server] Experiment data manager is designed to hold all of the metadata of an experiment in one object. This class stores the separate pieces of metadata in individual strings. The only methods the experiment data manager class contains are simple get and set methods to set and retrieve information within the class.

Experiment metadata implementation class is implemented using a variety of technologies. It uses JPanel from the swing package to construct the graphical user interface that is presented to user to enter the metadata to their experiment. This interface retrieves the metadata and sends it to be saved to persistent storage. The metadata is organized by being placed into an instance of the experiment metadata manager class. This instance of the experiment metadata class is then written to file in xml by using the XStream package [**?**]. The location of the metadata storage is the same as the experiments location for storing the stderr and stdout. This location is in a directory that shares the same name as the experiment it is associated with. The name of the experiment is ultimately determined not by the user but by the auto-naming method inside of the experiment metadata implementation class. This method automatically increments an integer that is attached to the end of the name of the experiment the user enters. To keep track of all of the experiments that have been created, experiment metadata implementation uses a vector that contains the string name of all of the experiments that have been added. This vector is the stored to persistent storage in xml format using XStream in the file entitled versions.xml. This file is referenced when the auto-naming method is determining the correct number to attach to the end of the name entered by the user.

Experiment output manager is designed to handle the user's query based commands. It uses XStream load the vector stored in versions.xml as well as other saved instances of the experiment data manager class. Once the necessary information has been loaded experiment output manager conducts the specified query through the data and returns the results.

5.2 Server

The server consists of four threads of execution; the main thread of execution listens for and responds to requests from the client, one for intermittent checkpointing of the queues, and another two threads to

update the status of submitted experiments.

The main thread instantiates the experiment manager class that is responsible for providing all the necessary methods that exposes the interface for the client to communicate with the server. Once the checkpointed queues have been loaded from the experiment path, two new threads are spawned, and they are responsible for updating the status of experiments in the submitted queue. The number of threads and their polling intervals can be adjusted for performance fine-tuning. The choice of using a couple threads to monitor experiment status allows the experiment management system provide reasonable response time while restricting resource consumption. These threads also automate the retrieval of any output associated with the experiment. Because they can detect an experiment's change in state, these threads update the standard output, error, and status on a minimal basis and conserving computational consumption. The final thread initiated during the experiment manager startup process saves the states of the four status queues at a configurable interval. The functionality addresses the possibility of an abrupt interruption preventing the experiment manager from gracefully halting.

Another thread is initiated when the experiment manager server is started. It uses a configuration polling duration to checkpoint the queues, so that an abrupt interruption will not cripple the system, and be able to launch the experiment management system with minimal loss.

The server uses four levels of class containment, and Figure 3 illustrates the containment relationship. At the root of the containment relationship is the ExperimentManager class that provides the server-side functions to respond to the commands that the client issues. The commands supported by the ExperimentManager class include add, submit, list, status, and stop. The ExperimentManager class implements the functions based on the four queues that it maintains: pending, submitted, completed, and failed. The queues consist of objects that hold an experiment data structure along with the experiment's id and a dependencies list.

The Experiment class is the core data structure in mediating communication between the experiment management system and the grid. The class exposes an interface to simplify the experiment submission process that incorporates enhanced status reporting through the standard status. The standard status is simply a file with entries describing the current status of an experiment that is written to the working directory where the executable program is invoked. The standardized format of each entry, as shown in Figure 4, permits customized status reporting. However, those applications used must be rebuilt to append to the standard status additional information about its execution state. The standard status, as currently implemented, reports the latter three states of the experiment state diagram: running, completed, and failed. It also supports more detailed error detection by reporting trapped signals that otherwise would have vanished on the remote host.

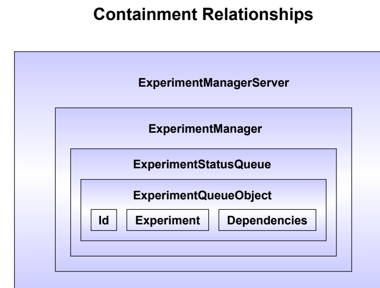

Figure 3: Containment relationship among the primary classes used to implement the experiment manager server

```
#CoG: <status> : <time> <date> <time_zone>

where
status ::= pending | submitted | running | completed | fai
time ::= HH:MM:SS
date ::= MM/DD/YYYY
time_zone ::= GMT
```

Figure 4: HSpecification of an entry for the standard status

The experiment class uses the Java CoG Kit's Task and FileOperation abstractions to interact with the grid. Preparation of the experiment for submission requires wrapping the executable and its the associated list of arguments into a shell script. This implementation of the standard status requires both the transfer of and setting execute permissions for the script on the remote machine. The experiment is

submitted through a task handler that provides simple status reporting via through the Status interface defined in the Java CoG Kit abstractions-common module. Automated checkpointing of the experiment occurs when the Experiment object detects that the experiment has been successfully submitted to the remote host for execution through a status change event notification.

6 Security Issues

As we have a simple client-server implementation, we require that the experiment client and server be run as part of a secure Intranet. However, as we have already implemented the logic of dealing with experiments, it will be straightforward to use secure grid sockets that are provided by the Java CoG Kit or Secure Grid Services provided by the Globus Toolkit 4. In both cases, the communication between the client and server can be securely achieved. At this time we provide a secure solution as both client and server can be run on the user's computer. Naturally, the communication between client and server is done through a port that is not externally accessible.

7 Performance Results

We logged the amount of memory and clock time required for the four primary server operations of adding, submitting, listing, and displaying the status of experiments under increasing loads. The number of experiments managed by the experiment manager provides the basis of measuring the load on the system. Memory consumption is an important indicator in determining how well the experiment management system will perform with other programs running concurrently. On a Pentium 4, 1.8 GHz machine running the Linux-2.4 operating system with 512 MB of memory, we obtained the following results pertaining to the amount of allocated heap space used by the experiment management system. Because of the fluctuating heap size allocated by the JVM, using the absolute byte count of used memory is not useful. Instead we take the percentage of the amount of heap used versus the total heap size. The results of the heap usage performance test, as summarized in Figure 5, show that the experiment manager consumes within the range of 75-80 percent of the available heap space when a reasonably large number of experiments have been submitted. The percentage is within the context of absolute allocated memory ranging from 2MB to 60 MB. The 60 MB requirement suggest that managing a thousand experiments can be problematic within a computing environment where memory is a scarce resource. On the other hand, the nearly constant 75-80 percent heap space consumption is a testament to the system's spatial scalability.

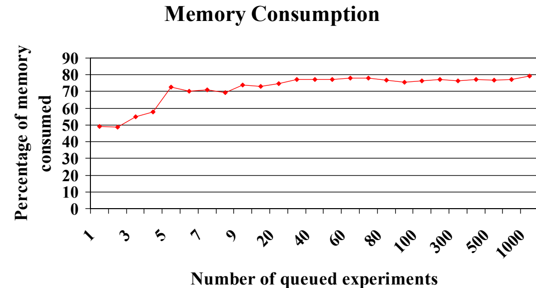

Figure 5: Percentage of allocated heap space consumed under increasing load of added experiments

In the time domain, the logs indicate a constant response time to requests to add, submit, and list the status of queue experiments. Adding experiment uses on average a range of 10 ms to 100 ms to satisfy the request. Checking the status of experiments requires on average 0.5 to 1.5 milliseconds to complete. Experiment submission requests on average approximately 0.5 to 1.5 seconds to locate the experiment, prepare the task for submission, and submit it. However, the amount of time needed to start the experiment manager requires orders of magnitude more time than the other operations. Beyond the 50 experiments threshold, we clocked an average of approximately one second per experiment, displaying linear growth performance. Below that threshold, the loading time grows linearly but at a rate that is less than 0.5 seconds per experiment. Figure 6 shows the difference in loading time below and above the threshold number of experiments. The results show that the system scales well when running; however, because of sizeable I/O involved in deserializing the checkpointed queues, reloading the experiment management system is an expensive operation. As such, it is advisable to categorize sets of experiments into separate projects, and manage the different categories of experiments as separate sessions.

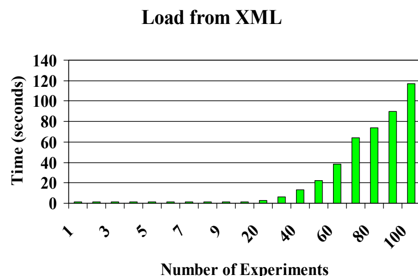

Figure 6: Amount of time to load checkpointed queues with an increasing number of experiments.

8 Conclusion

While looking at bioinformatics applications, we have identified that typical experiments need to be managed by the novice grid user. In order to support this requirement we have developed a tool called cog-experiment. This tool is architected around a client-server model that allows the user to manage a large number of tasks as part of his daily research quest. The experiment framekwork is bassed on a layered architecture that integrated fully with the Java CoG Kit. Through this combination we have enabled a system that allows automated checkpointing, automatic version control, and output file management. These features are making the researchers' experience to use the Grid simpler, faster, and more efficient. The cog-experiment is implemented using Java, and Java Cog Kit to provide these features.

Future research will focus on integrating our client-server model into a WS-RF based grid environment. Additionally, it will be simple to integrate our system into the Java CoG Kit's workflow framework.

9 Acknowledgements

This work was supported by the Mathematical, Information, and Computational Science Division subprogram of the Office of Advanced Scientific Computing Research, Office of Science, U.S. Department of Energy, under Contract W-31-109-Eng-38. DARPA, DOE, and NSF support Globus Project research and development. The Java CoG Kit Project is supported by DOE MICS, and NSF Alliance. This work was also supported by the National Science Foundation under Grant No. 0353989.

References

[1] G. von Laszewski and K. Amin, *Grid Middleware*. Wiley, 2004, ch. Middleware for Commnications, pp. 109–130. [Online]. Available: http://www.mcs.anl.gov/~gregor/papers/vonLaszewski--grid-middleware.pdf

[2] I. Foster, "What is the Grid? A Three Point Checklist," 22 July 2002. [Online]. Available: http://www.gridtoday.com/02/0722/100136.html

[3] G. von Laszewski, I. Foster, J. Gawor, W. Smith, and S. Tuecke, "CoG Kits: A Bridge between Commodity Distributed Computing and High-Performance Grids," in *ACM Java Grande 2000 Conference*, San Francisco, CA, 3-5 June 2000, pp. 97–106. [Online]. Available: http://www.mcs.anl.gov/~gregor/papers/vonLaszewski--cog-final.pdf

[4] G. von Laszewski and M. Hategan, "Grid Workflow - An Integrated Approach," in *To be published*, Argonne National Laboratory, Argonne National Laboratory, 9700 S. Cass Ave., Argonne, IL 60440, 2005. [Online]. Available: http://www.mcs.anl.gov/~gregor/papers/vonLaszewski-workflow-draft.pdf

A Command cog-experiment

```
NAME
        cog-experiment

SYNOPSIS

     cog-experiment [-info] [-list] [-tree]
                    [-add <experiment> [-file <filename>]]
                    [-submit <experiment>]
                    [-status <state>]
                    [-delete <experiment>]
                    [-combine <experiment1> <experiment2>]
                    [-metaset <experiment>]
                    [-copy <experiment> <directory>]
                    [-rename <old experiment> <new experiment>]
                    [-search <options> <string>]
                    [-diff <options> <file>]
                    [-view <options>]
                    [-edit <file>]
                    [-ls]

     A short format is also available:

     cog-experiment [-i] [-l] [-t]
                    [-add <experiment> [-file <filename>]]
                    [-submit <experiment>]
                    [-stat <state>]
                    [-del <experiment>]
                    [-com <experiment1> <experiment2>]
                    [-ms <experiment>]
                    [-cp <experiment> <directory>]
                    [-rn <old experiment> <new experiment>]
                    [-srch <options> <string>]
                    [-diff <options> <file>]
                    [-view <options>]
                    [-ls]


DESCRIPTION

    cog-experiment manages experiments.  Experiments can be added,
    deleted, submitted, and monitored.  Other management tasks include
    querying for the status of experiments and searching and filtering
    experiments. The stdout and stderr of each of experiment is
    managed using a directory structure that is customizable through
    the COG_EXPERIMENT_PATH environment variable.  An example for a
    directory structure for an experiment could look something like
    this: $HOME/.globus/experiment/<experiment name>/stdout /stderr

OPTIONS
        -info
                Returns useful information about the location of the
                experiment

        -list
                    Lists all experiment names

           -tree
                    Lists tree view of all experiment names

        -edit <filename>
                Edits file using emacs

           -add <experiment name>
```

```
            adds a new experiment without submitting it
        user prompted to enter experiment information

-add <experiment name> <option>
        add options:
        -file [filename]
                Uses the specified file as the task for the
                experiment

   -status
             lists the status of all experiments

   -status <submitted | pending | running | done>
               lists all experiments with the specified state

-submit
            creates new experiment in default directory

-submit <experiment>
        creates a new experiment named experiment or continues
            experiment

-delete <experiment>
             deletes the names experiment

-combine <experiement1> <experiment2>
             adds the data of experiment2 to the end of experiment1

-metaset <experiment>
             set the metadata to the named experiment

-settask <experiment>
        set the task on the named experiment

-copy <experiment1> <folder>
             copies experiment1 to destination folder

-rename <experiment1> <String n>
             renames experiment1 version nameto n

-search <search string> <option1> <option2>
             search option1:

                -dir <directoryname>
                        Executes function in specified directory

                -file <filename>
                        Executes task in specified file

        search option2:

                -metadata
                        Executes function returning only metadata

                -data
                        Executes function returning only the data

                -stderr
                        Executes function returning only
                        stderr

-diff <filename1>, <filename2>


-diff <filename1>, <filename2> <option>
```

```
        diff option:

                     -metadata
                        Executes function returning only metadata

                 -data
                        Executes function returning only the data

                 -stderr
                        Executes function returning only stderr

    -versions <experiment name>
                  lists the name of all files in the specified
                  experiment folder

    -experiments
                   lists all experiments

    -view <filename>
                   prints all information in named file to screen

    -view <filename> <option>
            view option:

                 -metadata
                          Executes function returning only
                          metadata

                 -data
                        Executes function returning only data

                 -stderr
                        Executes function returning only stderr
    -ls
            view advanced information on all experiments
```

EXAMPLES

In this eaxample we show the creation of a new experiment.

```
cog-experiment -edit blastrun.xml -format schema.xsl
```

Through a swing interface the user is asked to fill out the
following form.

```
Metadata:

# Identity #

Identifier:
Experiment Name: blastrun

# Contact #

Author: Pan
Department: MCS
Project:  Gene Sequencing
Phone:  630-252-1682
E-mail:  zimny@mcs.anl.gov
Date:  July 11, 2005
Time:  2:00PM

# Execution #

Program:  /home/user/simulation
```

```
Arguments: "-s 2 -i 20"

# Host

Directory:
Stdout: cog-stdout
Stderr: cog-stderr
Projectaccountnumber: <your tera grid account number>
Parameters:
        -p gt2
        -s tg-grid1.uc.teragrid.org/jobmanager-pbs
```

Now that the necessary information has been gathered, the experiment object is now created:

```
cog-experiment -add blastrun.xml
```

Next we send the experiment off to be completed on the grid:

```
cog-experiment -submit blastrun
```

If the experiment was run more than once the versions would look as follows:

```
cog-experiment -view -versions blastrun

blastrun-1
blastrun-2
...
```

To view the specific information of a experiment that has been completed:

```
cog-experiment -view -file blastrun-1

Metadata:
# Identity
Identifier:
Experiment Name: blastrun
# Contact:
Author: Pan
Department: MCS
Project:  Gene Sequencing
Phone:  630-252-1682
E-mail:  zimny@mcs.anl.gov
Date:  July 11, 2005
Time:  2:00PM
# Execution
Program:  /home/user/simulation
Arguments: "-s 2 -i 20"
# Host
Directory: ~
Stdout: cog-stdout
Stderr: cog-stderr
Projectaccountnumber: <your tera grid account number>
Parameters:
        -p gt2
        -s tg-grid1.uc.teragrid.org/jobmanager-pbs


Data:
12
23
43
56
76
```

```
Standard Error:
none

To view only the data created during the experiments run:

cog-experiment -view -file -data blastrun-1

Data:
12
23
43
56
76

cog-experiment view file data blastrun-2

Data:
12
23
34
56
76

cog-experiment -diff -data genes-1 genes-2

        blastrun-1        blastrun-2
line 3: 43                34

If more than one experiment has been created, multiple experiments
can by viewed by:

cog-experiment -view

Experiments:
default
genes
dna

If you would like to create an experiment from an existing XML file:

cog-experiment -add -file /home/experiments/lysosome.xml

Experiment lysosome has been created.

To view all experiments:

cog-experiment -view -experiments

default
blastrun
dna
lysosyme

To view more information on all experiments:

cog-experiment -ls

                UserName        Time                Date
blastrun:        Pan                2:00PM                July 11, 2005
dna:                Tan                11:01AM                July 22, 2005
lysosome:          Gregor                5:45AM                July 23, 2005


cog-experiment -cureCancer
```

```
Cancer is now cured!

SEE ALSO

cog-job-submit, cog-checkpoint-submit, cog-checkpoint-status
```

B Design

```
--------------------------------------------------------------------------------
Client <--> Experiment Manager <--> Grid infrastructure

Client (UI)
    - command-line utility
    - GUI (monitor and visualize output)

Experiment Manager
    - 4 Queues (pending, submitted, completed, failed)
    - Persistent storage
        * load/save queues from/to persistent storage
        * experiments repository
    - Scheduler
        * selects the next task to submit to the Grid, based on a scheduling policy
          possible policies: FIFO, random, priority-based
    - task submission:
        * submits task(s) to Grid through a workflow
        * create as a node in the workflow a wrapper script to detect
          completion or failure of the task
        * update the appropriate queues
        * update the experiments repository with stdout, stderr and any other
          output

How different components of our system interact:

Component1 <- interface between the components -> Component2

UI <- ExperimentManagerInterface -> Queues <- ExperimentSchedulerInterface ->
Scheduler <- workflow -> Grid
h
UI <- ExperimentRepositoryInterface -> ExperimentRepository
ExperimentRepository <- ExperimentRepositoryInterface -> ExperimentManager

--------------------------------------------------------------------------------
Interfaces
--------------------------------------------------------------------------------
interface StatusQueue
    * StatusQueue is used as the underlying data structure in ExperimentManager
--------------------------------------------------------------------------------
    /*
     * NOTE: StatusQueue implementations MUST be THREAD-SAFE * queue
     * object: [id, experiment name, task, dependencies]
     */

    /*
     * inserts an object into the queue returns a String that is the
     * id of the queued task
     */
    String enqueue(Object queueObject)


   /*
    * inserts an object into the queue with a list of dependencies
    * (dependencies is a list of ids that the added task depends on)
```

```
     * returns a String that is the id of the queued task
     */
     String enqueue(Object queueObject, List dependencies);

     /*
     * removes the next queue object from the queue, using the
     * scheduling policy returns the removed queue object
     */
     Object dequeue();

     /*
     * removes the queue object with the specified id
     * returns the removed queue object
     */
     Object dequeue(String id);

     /*
    * sets the dependencies for the queued task with the specified id
    */
     void setDependencies(String id, List dependencies);

     /*
    * sets the scheduling policy for the queue
    */
     void setSchedulingPolicy(String schedulingPolicy);

     /*
    * retrieves the scheduling policy for the queue
    */
     String getSchedulingPolicy();

     /*
    * enumerateQueueObjects returns an enumeration of queued objects
    */
     Enumeration enumerateQueueObjects();

--------------------------------------------------------------------------------
interface ExperimentManager
     * The client (user interface) interacts with the Experiment Manager through
       this interface
     * This interface is also used to interface with the Grid
--------------------------------------------------------------------------------
     /*
     * addTask adds tasks to an experiment (creates experiment
     * directory if necessary), and returns an array of Strings
     * representing task IDs that the pending queue has assigned to
     * each task.
     */
     String[] addTask(String experimentName, Vector tasks);

     /*
    * submitTask moves the task with the associated id to the submitted queue
    * The task gets executed according to the queue scheduling policy
    */
     void submitTask(String id);

     /*
    * enumerateTasks returns an enumeration of queue objects
    * queue object:  [id, experiment name, task, dependencies]
    * this function could be used to discover a task's id (as seen by the queues)
    */
     Enumeration enumerateTasks()

     /*
    * enumerateTasks with the status argument enumerates only queue
```

```
   * objects from a specified status queue.  queue object: [id,
   * experiment name, task, dependencies]
   */
    Enumeration enumerateTasks(String status);

    /*
   * getTaskStatus returns the status of the task with the associated
   * id Status is determined by which queue the task is located
   */
    String getTaskStatus(String id);

    /*
   * setTaskStatus moves the task with the specified id to a different
   * queue; the new queue is specified with the newStatus argument
   * intent of this method: to update a task's status after it has
   * been submitted; that is, when we get an update as to whether the
   * task has actually been completed or failed.  Determining
   * completion or failure is a sticky issue that we are currently
   * dealing with.
   */
    void setTaskStatus(String id, String newStatus);

    /*
   * only allow get methods to the queues
   * The user setting the queues causes inconsistency in status
   */
    StatusQueue getPendingQueue();
    StatusQueue getSubmittedQueue();
    StatusQueue getCompletedQueue();
    StatusQueue getFailedQueue();

    /*
   * writes the contents of all four queues out to persistent storage so that we
   * can recover from interruptions (such as the ExperimentManager being
   * stopped).
   */
    void checkpointStatusQueues();

    /*
   * We need some way to update the experiments repository upon receiving
   * information (such as stdout, stderr, or any program output files) from the Grid
   */

    /*
   * Writes stdout, stderr, or output files that need to be updated to the
   * experiment repository.  updateExperimentRepository updates the contents
   * based on a task's experiment name and id number?
   */
    void updateExperimentRepository()
    void updateExperimentRepository(String id)

--------------------------------------------------------------------------------
interface ExperimentScheduler
    * Interface for experiment manager to schedule and submit tasks to the Grid
    * this component schedules tasks for execution on the Grid based on a
      scheduling policy
--------------------------------------------------------------------------------
    /*
   * Given a list of tasks, determine which to go next
   * Possible scheduling schemes:  FIFO, random, priority
   * nextTask takes an enumeration and a scheduling policy and determines what
   * object in the enumeration is next
   */
    Object nextTask(Enumeration objectEnumeration, String schedulingPolicy);
```

```
--------------------------------------------------------------------------------
interface ExperimentRepository
    * Interface between the client and the experiment repository
    * Interface between the experiment manager and the experiment repository
--------------------------------------------------------------------------------
    /*
   * The experiment repository should maintain some form of checkpointing, so
   * that queries on it will simply be a matter of reading from the checkpoint
   * file instead of traversing the entire structure.
   * checkpointEperimentRepository saves the directory tree of the experiment
   * repository
   */
    void checkpointExperimentRepository();
    void setCheckpointFileName(String checkpointFileName);
    String getCheckpointFileName();

    /*
   * writes outputFile to the location where the experimentName is located
   * writeFlag argument determines whether to append or overwrite if the file
   * already exists.
   */
    void writeToRepository(String experimentName, File outputFile, int writeFlag);

--------------------------------------------------------------------------------
    * The interface between the experiment manager as a whole and the Grid is
      probably through workflows?
--------------------------------------------------------------------------------
```