# The Java CoG Kit Experiment Manager

Gregor von Laszewski,[1,2] Phillip Zimny,[3,1] Tan Trieu,[4,1] David Angulo[5]

[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] Loyola University Chicago, Department of Computer Science, Lewis Towers,
Suite 416, Water Tower Campus, Chicago, IL, 60611

[4] Santa Clara University, Department of Computer Engineering,
500 El Camino Real, Santa Clara, CA 95053

[5] DePaul University, School of Computer Science, Telecommunications, and Information Systems,
CTI 650, 243 S. Wabash, Chicago, IL 60604

## Abstract

In this paper, we introduce a framework for experiment management that simplifies the user's interaction with Grid environments. We have developed a service that allows the individual scientist to manage a large number of tasks as typically found in experiment management. Our service includes the ability to conduct application state notifications. Similar to the definition 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 shown 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 [1]. The analogy between the computational grid and the power grid highlights the emphasis on virtualization. A user plugs an appliance into the power outlet and 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 [2]. 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 promises 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 [2]. The implementation of the services form the Grid middleware, and the Globus Toolkit [3] is today's de facto standard Grid middleware [4]. The toolkit provides an elementary set of facilities to handle security, communication, information, resource management, and data management services [2]. 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 [2].

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. In addition, it contains a number of convenient services not found in Grid middleware. This means it not only defines a set of convenient classes that provide the Java programmer with access to basic Grid services [5], but also integrates a number of sophisticated abstractions, one of which is a workflow system [6]. Hence, it provides a significant feature enhancement to existing Grid middleware [2].

A popular use of the Grid is motivated by the field of bioinformatics, where applications such as Grid-enabled Basic Local Alignment Search Tool (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 submitted 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 also includes specific enhancements for the Grid environment. To emphasize the similarities, consider a typical use case of a user working in a UNIX command shell. The user queries which jobs have been submitted with the shell's history function. The status of the 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, as an extension to the ps command and history management tools provided by the shells, we must integrate user accessible outputs and error files on a command-by-command basis.

A preliminary version of the execution manager has been 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 integration of fault tolerant queues for managing experiment submissions.

The rest of the paper is structured as follows. First, we present a use case for our framework. Then we discuss the requirements derived from the use case that lead us to redesign the Grid Command Manager for experiment support. 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 Use Case

While motivation for this project was derived from several different use cases, this paper focuses on the following single use case of bioinformatics research.

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 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 biological function of the sequence they are studying by comparing it to similar known sequences BLAST finds [?].

Biologists may find themselves in the scenario where they wish to research a specific genetic sequence by making 500 different modifications to the sequence, and then run it through BLAST to see if the modification produces any characteristics of other sequences. The biologist would 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 and then keep track of the 500 different names. Upon the completion of the BLAST run, the biologist would then have to move the output files into separate directories which would have to be named and then remembered.

Once the biologist has completed the 500 BLAST runs, there are 500 different outputs to manage. The biologist most likely does not want to devise a method of organizing all of the different output they have created. Even once organized, there is little additional data associated with the outputs to allow the biologist to search through the output files. This research method is inefficient and time-consuming. In essence, it is not an optimal way for a biologist to conduct experiments.

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 many times necessary, while making slight modifications to the submission parameters. For this example, the modification to the parameters would be the name of the file in which the modified sequence was stored. 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 offering 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 assign a sequential version number to each submission. The biologist now has an easy to understand version schema. This auto-naming feature takes away any concerns 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 on the user's local machine. Now the biologist has all of their submission files properly named and has the output files neatly stored in individual folders. However, as this may still be too cumbersome to effectively manage all of the output, we introduce metadata into the experiment functionality.

The biologist can use the option to enter metadata about experiments on an individual basis. Information about a specific experiment such as author, time, or other notes can be saved to persistent storage. When biologists do this, it allows them to search more specifically through all the outputs. For example, if the biologist wants to view all the experiments generated from a certain day, a simple search can be done for that date which displays all experiments from that date to the screen.

## 3 Requirements

From the use case described, we derive several major requirements that include automated experiment checkpointing, transparent output management, automated version control, metadata management, application status reporting, persistent experiment sessions, and scalable experiment updating. Next, we will discuss each of the requirements in 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. After submitting an experiment, a user must only associate the experiment with its name in order to track its status.

**Transparent Output Management.** To shield the user from details about the Grid, the standard output (stdout) and standard error (stderr) 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 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 future comparison. Automated version control removes the responsibilities of renaming, moving, and organizing different versions of output from the scientist.

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

**Application Status Reporting.** Besides retrieving stdout and stderr, we believe that users will benefit from application status reporting. When checking the status of an experiment that has failed the user may wonder what triggered or caused the failure. The user can query a standard status to review the events that occurred before the failure.

The use of a 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.

**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 update the status of all experiments in a scalable fashion.

## 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 [6].

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 process 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 that periodically updates the status of the queued experiments.

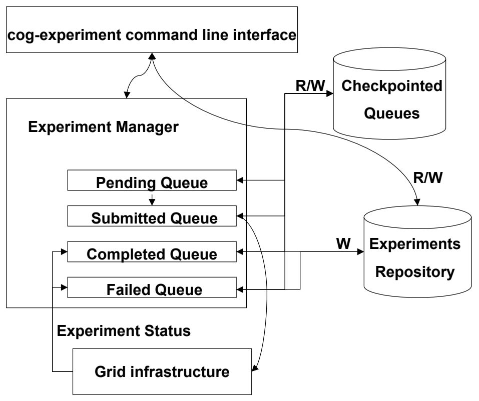

Figure 1: The architecture of the Java CoG Kit experiment management framework.

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 an experiment. To provide persistent experiment ses-

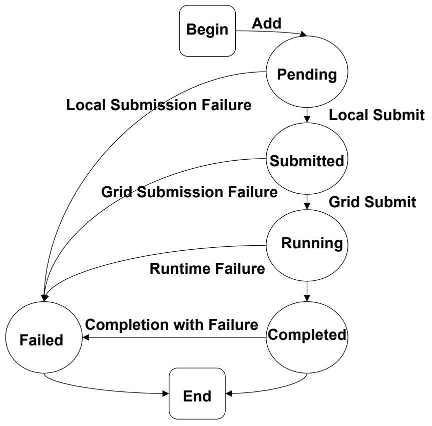

Figure 2: State diagram of an experiment as it transitions through the four queues.

sion functions, the experiment manager periodically checkpoints the status of the four queues to a 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 read and write access to the experiments repository.

## 5 Implementation

The implementation of the experiment management system is split into two components: the client command line interface and the experiment manager service.

### 5.1 Client

The client is composed of four key classes, ExperimentManagerClient, ExperimentMetadataImpl, ExperimentOutputManager, and ExperimentDataManager.

The ExperimentManagerClient class parses the command line arguments provided by the user to determine the appropriate actions to take. ExperimentManagerClient's file management methods such as auto-versioning or metadata storage use instances of the ExperimentMetadataImpl class. The view and search methods of ExperimentManagerClient use instances of the ExperimentOutputManager class. To help keep the metadata organized throughout the client, ExperimentMetadataImpl and ExperimentOutputManager both use instances of the class ExperimentDataManager. This class hierarchy is visualized in Figure 3.

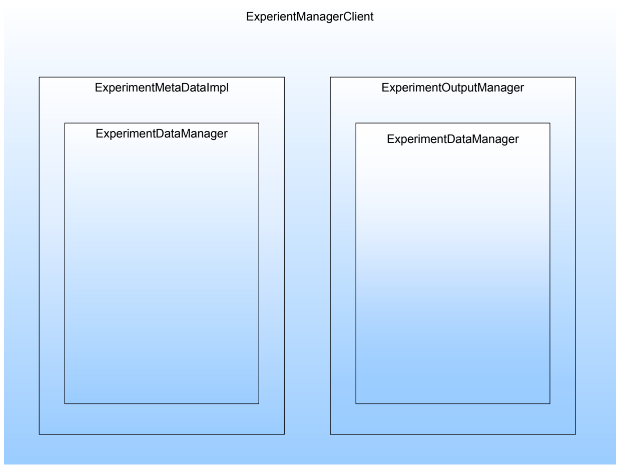

Figure 3: Visualization of the client class hierarchy.

ExperimentDataManager manages all of the metadata of an experiment and allows the client to reference the metadata through the instance of one object. This class stores the separate pieces of metadata. The ExperimentDataManager class contains only simple methods such to set and retrieve information from within the class.

ExperimentMetadataImpl class is implemented using a variety of technologies. It uses JPanel from Java's SWING package to construct the graphical user interface that is presented to the user for entering the metadata for an experiment. This interface retrieves the metadata, stores the information as an instance of the ExperimentDataManager class, and saves the instance to persistent storage. The instance of the ExperimentDataManager class is written to an XML file named after the experiment using the toXML() method from the XStream package [?].

The location at which experiment files are stored to persistent data can be customized by setting the COG_EXPERIMENT_PATH environment variable to a user specified location. Upon creating an experiment, a folder named after the experiment is created to store the experiment's metadata file, stdout, stderr, and stdstatus. Figure 4 depicts the default COG_EXPERIMENT_PATH directory hierarchy.

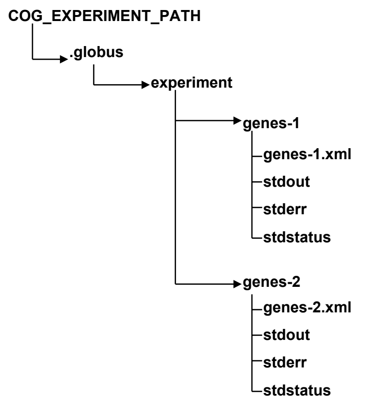

Figure 4: Default COG_EXPERIMENT_PATH directory hierarchy.

As one of the default settings, the name of the experiment is ultimately determined not by the user, but by the auto-naming method inside of the ExperimentMetadataImpl class. This method automatically increments a number that is attached to the end of the name of the experiment the user enters. To keep track of all the experiments that have been created, ExperimentMetadataImpl writes all of the experiment names to persistent storage. These names are saved to an XML file named versions.xml using the XStream toXML() method. This file is referenced when the auto-naming method determines the correct version number for an experiment or when the client needs to know the names of all the experiments.

ExperimentOutputManager is designed to handle the user's view and search commands. It uses XStream's fromXML() to load the names stored in the versions.xml file as well as each experiment's saved instance of the ExperimentDataManager class. Once the necessary information has been loaded, experiment output manager conducts the specified command and returns the results.

## 5.2 Server

The server consists of four threads of execution: (1) the main thread of execution listens for and responds to requests from the client; (2) another thread intermittently checkpoints the four queues; (3) and (4) two threads update the status of submitted experiments.

The main thread instantiates the experiment manager class that is responsible for providing all the necessary methods to expose an interface for the client to communicate with the server. Upon startup, the main thread loads the checkpointed queues, if any, from the experiment path that is specified by the COG_EXPERIMENT_PATH environment variable. Once the checkpointed queues are loaded, the main thread spawns two new threads that 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. Using only two threads to monitor and update experiment status allows the experiment management system to 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. Thus the use of the two threads, instead of creating a new thread for each experiment, helps conserve compute cycles and reduce disk I/O. The final thread initiated during the experiment manager startup process saves the states of the four status queues at a configurable interval. Periodically checkpointing the four queues addresses the possibility of an abrupt interruption preventing the experiment manager from gracefully halting.

The server uses four levels of class containment, and Figure 5 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 environment. The class exposes an interface to simplify the experiment submission process that also incorporates enhanced status reporting through the standard status. We considered three methods of implementing application status notification. The first implementation we considered simply directs the notifications to the user. However, such an approach would undermine the underlying assumption that experiments are non-interactive and long running. The purpose of using

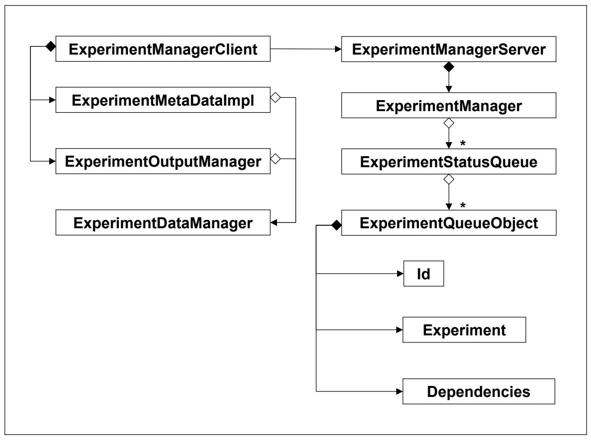

Figure 5: Containment relationship among the primary classes used to implement the experiment manager server.

the checkpointing mechanism provided by the Java CoG Kit's abstractions module is to address the overhead of maintaining a persistent network connection with the remote execution host. Losing the network connection means losing all status notifications. The second alternative to providing application status notifications is to append to the stdout. The problem with this approach is that the notifications could get buried when applications excessively write to stdout. The third method we considered defines the standard status as a separate file, similar to how stderr is separated from stdout in the UNIX environment. This definition addresses the shortcomings of the two previous alternatives: status notifications are saved despite losing network connectivity, and status messages are separated from the stdout.

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 6, permits customized status reporting. However, the applications that integrate runtime status notification must be rebuilt to append to the standard status additional information about the application's 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.

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

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

Figure 6: Specification of an entry for the standard status.

The experiment class uses the Java CoG Kit's task and file operation abstractions to interact with the Grid environment. Preparation of the experiment for submission requires wrapping the executable and its associated list of arguments into a shell script. This implementation of the standard status requires transferring the script to the remote host and then setting the execute permission 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.

## 6 Security Issues

With a simple client-server implementation, we require that the experiment client and server operate in a secure Intranet. However, as we have already implemented the logic for managing experiments, it will be straightforward to use either the Java CoG Kit's secure grid sockets or the Globus Toolkit 4's secure grid services. 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 run on the same machine, with the client and the server communicating through a port that is externally inaccessible.

## 7 Performance Results

We logged the amount of memory and wall 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 maintained by the experiment manager provides the basis of measuring the load on the system. Memory consumption is an important indicator of 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 RAM, 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 Java Virtual Machine (JVM), evaluating the absolute byte count of used memory is not useful. Instead, we analyze the percentage of the amount of heap space used versus the total heap size. The results of the heap usage performance test, as summarized in Figure 7, 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 added. However, the percentages exist within the context of the total heap space allocated by the JVM ranging from 2MB to 60MB for loads ranging from one experiment to 1000 experiments. The 60 MB needed to maintain 1000 experiments suggests that managing 1000 experiments can be problematic in an 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.

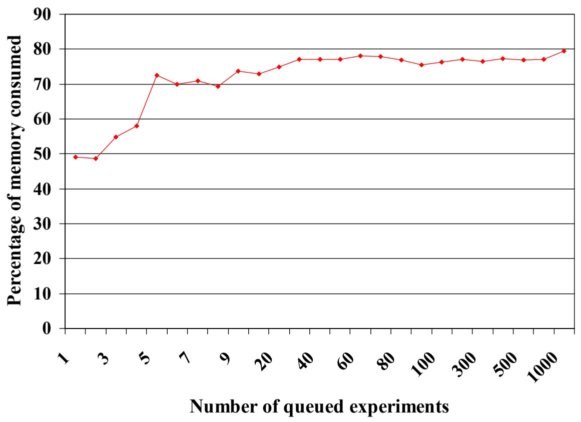

Figure 7: Percentage of allocated heap space consumed under increasing load of added experiments.

In the time domain, the logs indicate a constant response time for requests to add, submit, and list the status of queue experiments. Adding an experiment requires a range of 10 ms to 100 ms to execute. Checking the status of experiments requires approximately 0.5 ms to 1.5 ms to complete. Experiment submissions take approximately 0.5 to 1.5 seconds to locate the experiment, prepare the task for submission, and submit the task. 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 8 shows the difference in loading time below and above the threshold number of experiments.

The performance results show that the system scales well when running; however, because of sizeable disk 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 with separate sessions.

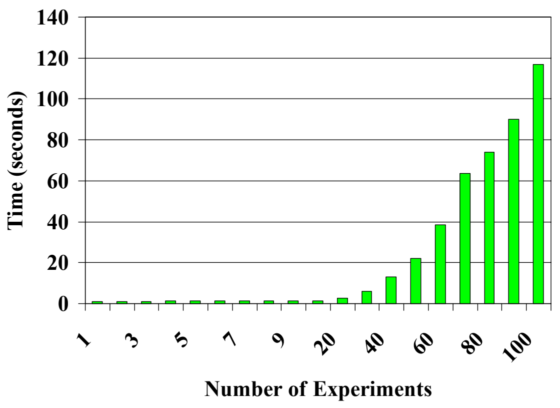

Figure 8: Amount of time to load checkpointed queues with an increasing number of experiments.

## 8 Conclusion

While reviewing 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 structured around a client-server model that allows the user to manage a large number of tasks as part of a daily research quest. The experiment framework is based on a layered architecture that integrates fully with the Java CoG Kit. With this combination, we have developed a system that incorporates automated checkpointing, automatic version control, and output file management. These features enable researchers to interact with the Grid in a simpler and more efficient fashion. The cog-experiment command line tool is implemented in Java, and uses the Java CoG Kit to provide the experiment management features.

Future research will focus on integrating our client-server model into a Web Services Resource Framework (WS-RF) [7] 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] I. Foster, "The anatomy of the grid: Enabling scalable virtual organizations," *International Journal of High Performance Computing Applications*, vol. 15, no. 3, pp. 200–222, August 2001, a brief introduciton to the grid. [Online]. Available: http://www.gl.iit.edu/database/frame/compendex.htm

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

[3] "The Globus Alliance," Web Page. [Online]. Available: http://www.globus.org

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

[5] 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

[6] G. von Laszewski and M. Hategan, "Grid Workflow - An Integrated Approach," in *Technical Report.*, 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

[7] "Web Services Resource Framework (WSRF)," Web Page. [Online]. Available: http://www.globus.org/wsrf

## A Command cog-experiment

```
NAME
    cog-experiment

SYNOPSIS

     cog-experiment[-help]
                   [-list]
                   [-add <experiment name> -service-contact <host>
                         -executable <file>
                         [-job-manager <jobmanager>]
                         [-provider <provider>][-arguments <string>]
                         [-enviroment <string>][-directory <string>]
                         [-batch][-redirected][-stdout <file>]
                         [-stderr<file>][-attributes<string>]
                         [-checkpoint <filename>][-verbose]
                         [-metadataoff][-versionoff]]
                   [-submit <experiment name> | all]
                   [-status <option>]
                   [-delete <experiment name>]
                   [-metaset <experiment name>]
                   [-search <string> [-dir <experiment>][-metadata]
                         [-data][-error][-stdstatus]]
                   [-view <experiment name> [-metadata][-output]
                          [-error][-stdstatus]]
                   [-stop]

     For some of the options a short form is available:

     cog-experiment[-h]
                   [-l]
                   [-add <experiment name> -s <host> -e <file>
                         [-jm <jobmanager>][-p <provider>]
                         [-args <string>][-env<string>]
                         [-dir<string>][b][-r][-stdout<file>]
                         [-stderr<file>][-a <string>][-c <filename>]
                         [-v][-mo][-vo]]
                   [-sub <experiment name> | all]
                   [-stat <state>]
                   [-del <experiment name>]
                   [-search <string> [-dir <experiment name>][-md]
                         [-out][-err][-stdstatus]
                   [-view <experiment name> [-md][-out]
                          [-err][-stdstatus]]
                   [-stop]


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.  A paper
    at (insert link) describes the cog-experiment framework in more
    detail.  Please read through the simple examples provided
    at the bottom before reviewing the descriptions of each option.

OPTIONS
       -help
            Returns information about the available commands
       -list
            Lists the name and experiment ID of all experiments

       -add <experiment name> -service-contact <host>
                         -executable <file>
                         [-job-manager <jobmanager>]
                         [-provider <provider>][-arguments <string>]
                         [-enviroment <string>][-directory <string>]
                         [-batch][-redirected][-stdout <file>]
                         [-stderr<file>][-attributes<string>]
                         [-checkpoint <filename>][-verbose]
                         [-metadataoff][-versionoff]]
            Creates a new experiment without submitting it
            user prompted to enter experiment information

            -service-contact
                specify service contact or the host to perform
                task

            -exectutable
                specifies executable file or the program used to
                conduct the computations for the experiment.  this
                is required when adding a job.

            -job-manager
                Execution job manager (fork, pbs, etc)

            -provider
                choose specific provider or type of middleware to
                use

            -arguments
                specify arguments used in the grid computation.
                If more than one use quotes

            -enviroment
                enviroment variable for the remote execution
                enviroment, specified as name=value[,name=vaule]

            -directory
                set the directory where information is sent to
                on remote machine

            -batch
                if present, job runs in batch mode or not
                interacting with user until completion

            -redirected
                if present, the arguments to -stdout and -stderr
                refer to local files

            -stdout
                indicates the file where the standard output of the
                job should be directed

            -stderr
                indicates the file where the standard error of the
                job should be directed

            -attributes
                additional task specification attributes.  attributes
                can be specified as "name=value[,name=value]"

            -checkpoint
```

```
          checkpoint file name. checkpointing or auto-recovery
          information is stored in this file

      -verbose
          if enabled, more information about what is
          being done is displayed

      -metadataoff
          when present disables the GUI to enter metadata

      -versionoff
          when present the auto versioning feature is disabled

 -status [state|name]
      Available states:

      all
          Lists all experiments in all queue

      pending
          Lists all experiments in pending queue

      submitted
          Lists all experiments in the submitted queue

      completed
          Lists all experiments in the completed queue

      failed
          Lists all experiments in the failed queue

      name:

      experiment name
          Displays status information of named experiment

  -submit <experiment name | all>

      experiment name
          submits experiment to be processed

      all
          submits all experiments waiting to be submitted

  -delete <experiment name>
          deletes the named experiment

  -edit <experiment name>
          edit metadata to named experiment

  -search <string>
          search for string in all experiment directories

  -search <string> [-dir <experiment name>][-metadata]
                    [-output][-error][-stdstatus]

          -dir <experiment name>
               searches specified experiment

          -metadata
               searches only metadata

          -output
               searches only stdout
```

```
                -error
                     searches only stderr

                -stdstatus
                     searches only stdstatus

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

        -experiments
                lists all experiments

        -view <experiment name>
                prints all information in the named experiment to screen

        -view <experiment name> [-metadata][-output][-error][-stdstatus]

                -metadata
                        displys experiment's metadata to screen

                -output
                        displays experiment's stdout to screen

                -error
                        displays experiment's stderr to screen

                -stdstatus
                        displays experiment's stdstatus to screen

        -stop

                stops server

Examples:

Standard -add command:

./cog-experiment -add genes -e /bin/date -s localhost

This will prompt the user to enter the metadata JFrame window:

--------------------------------
Name            genes-1
Author          Phill
Department      MCS
Project         genetics
Phone           555-555-5555
E-mail          cog@cogkit.org
Date            August 16, 2005
Time
Program Used    BLAST
Argument
Account Number
Parameters
--------------------------------
genes-1 has been added

Standard -list command:

./cog-experiment -list

This will list all added experiments:

genes-1
```

Second Standarad -add command:

```
./cog-experiment -add genes -e /bin/date -s localhost
```

This will prompt the user to enter the metadata JFrame window:

```
--------------------------------
Name            genes-2
Author          Phill
Department      MCS
Project         genetics
Phone           555-555-5555
E-mail          cog@cogkit.org
Date            August 17, 2005
Time
Program Used    BLAST
Argument
Account Number
Parameters
--------------------------------
genes-2 has been added
```

Standard -submit command:

```
./cog-experiment -submit genes-1
```

This will submit genes-1 to it's service contact:

```
genes-1 has been submitted
```

View metadata command:

```
./cog-experiment -view genes-1 -metadata
```

This will display genes-1 metadata:

```
Name            genes-1
Author                Phill
Department      MCS
Project         genetics
Phone           555-555-5555
E-mail          cog@cogkit.org
Date            August 16, 2005
Time
Program Used    BLAST
Argument
Account Number
Parameters
```

Standard -versions command:

```
./cog-experiment -versions genes
```

This will display all versions of genes:

```
Versions of genes:

genes-1
genes-2
```

Standard status command:

```
./cog-experiment -status genes-1
```

```
This will return the status of the genes-1:

[1] submitted 4000 genes-1
```

# B Interfaces

## B.1 Experiment Interface

```
/**
* Experiment is an interface to an experiment, which exposes a simple submit method
* The submission process requires preparation of a wrapper script that implements
* the standard status and the transfer of that script to the remote host for execution
*/
public interface Experiment {
    /**
     * getCurrentStatus returns the experiment's current status
     *          (in this case, in which queue the experiment resides)
     * @return a String from the ExperimentStates class that holds the experiment's
     *          status
     */
    public String getCurrentStatus();

    /**
     * setCurrentStatus updates the current status of the experiment
     * @param currentStatus the new status of the experiment
     */
    public void setCurrentStatus(String newStatus);

    /**
     * getExperimentStopWatch returns the stopwatch that records how long the
     *          experiment resides in each queue
     * @return a StopWatch that is used to maintain how long an experiment
     *          has sits in each of the queues
     */
    public StopWatch getExperimentStopWatch();

    /**
     * setExperimentStopWatch replaces the current stopwatch with the one specified
     * @param stopWatch the StopWatch
     */
    public void setExperimentStopWatch(StopWatch stopWatch);

    /**
     * saveStandardOutput saves the standard output to the specified destionation path
     * @param String the location to save the standard output
     */
    public void saveStandardOutput(String destinationPath);

    /**
     * saveStandardError saves the standard error to the specified destionation path
     * @param String the location to save the standard error
     */
```

```
public void saveStandardError(String destinationPath);

/**
 * saveStandardStatus saves the standard status to the specified destionation path
 * @param String the location to save the standard status
 */
public void saveStandardStatus(String destinationPath);

// The rest of these methods replicate the interface to the
// cog-job-submit interface
public Task getExecutionTask();
public void setExecutionTask(Task executionTask);

public void submitTask() throws Exception;

public String getArguments();
public void setArguments(String arguments);

public String getExecutable();
public void setExecutable(String executable);

public boolean isBatch();
public void setBatch(boolean batch);

public boolean isRedirected();
public void setRedirected(boolean redirected);

public String getProvider();
public void setProvider(String provider);

public String getStderr();
public void setStderr(String stderr);

public String getStdout();
public void setStdout(String stdout);

public void setCommandline(boolean bool);
public boolean isCommandline();

public String getCheckpointFile();
public void setCheckpointFile(String file);

public String getName();
public void setName(String name);

public String getServiceContact();
public void setServiceContact(String serviceContact);

public String getDirectory();
public void setDirectory(String directory);
```

```
    public String getEnvironment();
    public void setEnvironment(String environment);

    public String getAttributes();
    public void setAttributes(String attributes);

    public String getJobManager();
    public void setJobManager(String jobmanager);

}
```

## B.2 ExperimentQueueObject Interface

```
/**
* ExperimentQueueObject is an interface to an the object that gets queued into
* an experiment status queue
* Such an object should provide access to an experiment, that experiment's
* unique identifier, and any dependencies that an experiment may have
*/
public interface ExperimentQueueObject {

    /**
     * setId sets the unique identifier for an experiment in the status queue
     * @param String the new unique identifier associated with an experiment
     */
    public void setId(String id);

    /**
     * getId returns the unique identifier associated with the experiment contained
     *   in the current experiment object
     * @return String the unique experiment identifier
     */
    public String getId();

    /**
     * setExperiment sets the experiment associated with the queueObject
     * @param experiment the object to associate with the queueObject
     */
    // we should really replace the 'Object' references with Experiment references to
    // eliminate unnecessary casting, & thus improve performance
    public void setExperiment(Object experiment);

    /**
     * getExperiment returns the experiment associated with the queue object
     * @return the experiment associated with the queue object
     */
    public Object getExperiment();

    /**
     * setDependencies updates the dependencies list associated with an queue object
     * @param dependencies a Vector of other queue objects that this queue object
     *          depends on
     */
    public void setDependencies(Vector dependencies);

    /**
     * getDependencies returns the dependencies list associated with the queue object
     * @return a Vector containing the queue object's dependencies list
     */
    public Vector getDependencies();

}
```

## B.3 ExperimentStatusQueue Interface

```
/**
* ExperimentStatusQueue is the interface to the queue of experiments
*/
public interface ExperimentStatusQueue {
    // we should remove the Object references for enqueue and dequeue, and
    // replace them with ExperimentQueueObject
    /**
     * enqueue pushes an experiment into the queue using FIFO
     * @param experiment the object to enqueue
     * @return a String holding the enqueued object's id
     */
    public String enqueue(Object experiment);

    /**
     * enqueue pushes an experiment with the specified id into the queue using FIFO
     * @param experiment the object to enqueue
     * @return a String holding the enqueued object's id
     *
     */
    public String enqueue(Object experiment, String id);

    /**
     * enqueue pushes an object with the specified id and dependencies list in the
     *          queue in FIFO order
     * @param experiment the experiment to enqueue
     * @param id the id of the experiment to enqueue
     * @param dependencies the list of dependencies for the experiment to enqueue
     * @return a String holding the enqueued object's id
     */
    public String enqueue(Object experiment, String id, Vector dependencies);

    /**
     * dequeue removes the next object from the queue based on a scheduling policy
     * @return the object removed from the queue
     */
    public Object dequeue();

    /**
     * dequeue removes the object with the specified id from the queue
     * @param id the id of the object to dequeue
     * @return the object that has been dequeued
     */
    public Object dequeue(String id);

    /**
     * setDependencies sets the new dependencies list for the object with the given id
     * @param id the id of the object to change the dependencies list
     * @param dependencies the new dependencies list
     */
    public void setDependencies(String id, Vector dependencies);

    /**
     * getDependencies returns the list of dependencies for the object with the given id
     * @param id the id of the object to retrieve the depdencies from
     * @return a Vector containing the dependencies of the object with the given id
     */
    public Vector getDependencies(String id);

    /**
     * setSchedulingPolicy changes the scheduling policy to the one specified
     * @param schedulingPolicy the new scheduling policy
     */
    public void setSchedulingPolicy(String schedulingPolicy);

    /**
     * getSchedulingPolicy returns the current scheduling policy for dequeuing
     * @return a String holding the scheduling policy
     */
    public String getSchedulingPolicy();

    /**
     * enumerateQueue returns an array of all the objects in the queue
     * @return an array of objects in the queue
     */
    public Object[] enumerateQueue();
}
```

## B.4 ExperimentManager Interface

```
public interface ExperimentManager {
    /**
    * addExperiment adds a named experiment to the pending queue
    * @param experimentName the name of the experiment
    * @param experiment the experiment to add
    */
    public String addExperiment(String experimentName, Object experiment);

    /**
    * removeExperiment removes the experiment with the given id from the
    * experiment management system
    * @param id the id assigned to an experiment during the add process
```

```
*/
public void removeExperiment(String id);

/**
* submitExperiment moves the experiment with the given id from the pending
* queue to the submitted queue
* @param id the id associated with an experiment
*/
public void submitExperiment(String id);

/**
* enumerateExperiments returns an array of queue objects in all the queues
* @return an array of queue objects form all the queues
*/
public Object[] enumerateExperiments();

/**
* enumerateExperiments returns an array of queue objects from a particular
* status queue
* @param status the status (queue) to enumerate
* @return an array of queue objects with the given status
*/
public Object[] enumerateExperiments(String status);

/**
* getExperimentStatus returns the status of an experiment with the given id
* @param id the id associated with an experiment
* @return the status of the experiment of interest
*/
public String getExperimentStatus(String id);

/**
* setExperimentStatus moves an experiment with the given id to a new queue
* specified by newStatus
* @param id the id associated with an experiment
* @param newStatus the new status
*/
public void setExperimentStatus(String id, String newStatus);

public ExperimentStatusQueue getPendingQueue();
public ExperimentStatusQueue getSubmittedQueue();
public ExperimentStatusQueue getCompletedQueue();
public ExperimentStatusQueue getFailedQueue();

/**
* checkpointExperimentStatusQueues saves all the status queues to
* persistent storage in XML format; the checkpoint files are saved under
* the experiment's directory as hidden files
*/
public void checkpointExperimentStatusQueues();
```

```
    /**
    * updateExperimentRepository does an update on all files from all
    * experiments
    */
    public void updateExperimentRepository();

    /**
    * updateExperimentRepository writes out all files associated with the
    * experiment with the given id; files include stdout, stderr, stdstatus ...
    * @param id the id associated with an experiment
    */
    public void updateExperimentRepository(String id);

}
```