Extending the Cluster-Grid Interface Using Batch System Abstraction and Idealization

A. Nishandar, D. Levine, S. Jain
University of Texas at Arlington, Arlington, TX 76019, USA
{nishanda, levine}@cse.uta.edu, sankalp@fnal.gov

G. Garzoglio, I. Terekhov
Fermi National Accelerator Laboratory, Batavia, IL 60510, USA
{garzoglio, terekhov}@fnal.gov

Abstract

The grid computing technologies in use today provide simplistic interfaces to various batch systems that manage the clusters connected to a grid. These interfaces work fine for running simple applications but when complex applications such as High Energy Physics simulations are run on a grid, problems are exposed with these simplistic interfaces which make the integration of a cluster into a grid complex. In addition to this the grid middleware is not completely isolated from the batch systems. Thus in order to incorporate a new batch system into a grid, a new interface must be written for that batch system. This requires an understanding of the functioning of the grid middleware. Development and testing of these interfaces requires a lot of human effort. In this paper we identify some of the problems in integration of batch systems into a grid that are overlooked by current grid technologies and propose a framework which remedies these problems and enables the easy integration of clusters in a grid by providing a layer of abstraction between the grid middleware and the batch system managing the cluster.

1. Introduction

A typical grid environment consists of multiple clusters of computers running standard operating systems with additional middleware, for example Globus [1]. Each cluster may be administered by different organizations and may be controlled by different batch systems that have different interfaces for cluster job management. For a computational grid to incorporate different types of clusters, the grid middleware must support many types of batch systems or at least have the provision support new batch systems. For example in Globus this integration is done by writing a *jobmanager* for each batch system that is to be included in a grid. The jobmanagers implement simplistic interfaces to submit, kill and poll local jobs at a site.

A large amount of human effort is expended in the integration of computational resources with a grid middleware since enabling job management at a local site in a grid requires more sophisticated interfaces than just simple batch system interfaces. The rest of this paper is organized as follows: First an overview of cluster computing environment is presented. Then an overview of computational grids consisting of multiple batch systems is provided. Then we discuss problems that are encountered during integration of a batch system into a grid. Then we discuss the tools that we have developed to enable the integration of clusters with a grid middleware.

2. Background

A computational cluster is a group of computers that are connected together over high speed networks such as Gigabit Ethernet and work together as a unit to solve complex and computationally intensive problems. The computers that are a part of the cluster run standard operating systems such as Linux, Sun OS or Microsoft Windows and middleware to provide management of the cluster resources such as Portable Batch System [2], Condor [3], and Farms Batch System Next Generation [4]. The middleware that provide management of resources in a cluster is also called a *batch system*. A cluster has a homogeneous environment i.e. all the computers in the cluster have the same processor architecture, the same operating system and run the same cluster management middleware. A unit of computation on a cluster is called a *job*, where a job can be running an independent executable or it could be the part of a complex parallel application that has many jobs running on other nodes (computers) in the cluster.

The batch systems provide interfaces that let users submit, monitor and kill jobs. These interfaces are most often in the form of a command line interface and a programmer's interface. While submitting job(s) the user specifies the executable to run, the requirements of the job such as physical memory required by the job, required

computational time, any arguments to the executable and the path on local machine where the standard output and error files from the job's execution should be created. These requirements are submitted to the batch system server or scheduler, which then starts the execution of the job at one of the computers in the cluster. The machine from where the jobs can be submitted to the scheduler is called the *submit node* of the cluster and the machines where the actual computation takes place is called the *worker nodes*. Depending on the size of the cluster it may have multiple schedulers and multiple submit nodes.

To facilitate the execution of jobs at worker nodes a daemon process runs at each worker node. When a job is submitted to the scheduler it communicates with the daemon process at a worker node instructing it to launch the execution at that node. The scheduler then keeps track of the execution by periodically communicating with the daemon. The users can keep track of the jobs that they submitted by using the batch system commands for checking the status of jobs. Similarly one can kill jobs by using the batch system commands. Once a job completes the standard output and error files are returned to the computer and directory path specified at the time of submission. The way this job management is implemented in a cluster differs from one batch system to another, but in general execution of any batch system command results in some network communication between the batch system servers and the daemons running at the worker nodes.

Table 1: Different batch system commands

Command Type	PBS	CONDOR	FBSNG
Job Submission command	qsub <arg list> e.g. qsub	condor_submit <jdf_file>	fbs submit <jdf_file> *or* fbs submit <arg list>
Job Lookup command	qstat	condor_q	fbs lj
Job Kill command	qdel	condor_rm	fbs kill

Different batch systems have different interfaces/commands to allow user operations. Table 1 lists the command line interfaces for three batch systems – PBS, Condor and FBSNG. In PBS jobs can be submitted to the batch system using the *qsub* command which accepts a list of arguments to specify the path of the executable, requirements of the job, and arguments. In the Condor batch system, in order to submit a job the user must create a *job description file (jdf)* that contains the executable, its arguments and requirements. FBSNG accepts either command line arguments or a job description file for job submission.

3. Computational grids

A computational grid consists of many clusters of computers connected together by grid middleware such as Globus. Each cluster may be managed by a different batch system. Figure 1 shows a computational grid that has three batch systems connected to it – PBS, Condor and FBSNG. As shown, each cluster runs grid middleware in addition to the local batch system at the cluster. In addition, each cluster may be under separate administration.

The process of job submission to a grid is similar to that in a cluster. The user specifies the executable to run, its arguments, the requirements of the job, and the path on local machine where the standard output and error files from the job need to be deposited. It is the responsibility of the grid scheduler to find a resource (in this case a cluster) that meets user requirements and launch the execution there.

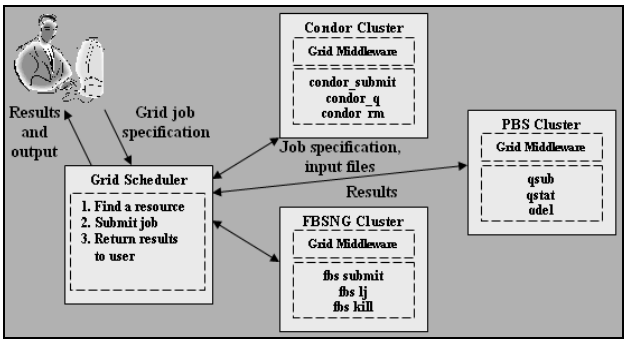

Figure 1: A computational grid with multiple clusters

It is worthwhile to note that the grid middleware provides only the *grid level job management*. The *local job management* at each cluster is still provided by the batch system running at that cluster. The grid level job management involves finding a cluster that is capable of running user job(s) and once such a resource is identified the grid scheduler needs to start the execution of the job at that cluster. This is done by submitting batch or local job(s) to that cluster using the batch system interfaces available there. In a grid environment the machine that is used as the submit node by the grid scheduler is called the *gateway node* or the *head node* of the cluster. The grid middleware also needs to keep track of the grid job status and once the job finishes return the output to the user. For this the middleware invokes the batch system interfaces and checks the status of the local jobs submitted as a result of the grid job.

The grid middleware completely isolates the batch systems from the grid user. The user does not even know which cluster is running the grid job. Thus the grid middleware needs to interface with different batch systems at different sites and give the user an abstraction of a homogeneous computing environment.

4. Problems with batch system integration in computational grids

When a batch system is used in conjunction with a grid, new problems are exposed which may be acceptable to a user who is using the batch system locally (an interactive user) but not to the grid middleware. Below we identify some of these problems and discuss how they effect the execution of jobs in a grid environment.

If a batch system command fails due to some reason, such as the command timing out or some other transient network failure, it will result in the grid middleware failing to execute the appropriate batch system interface correctly. For example, during the submission of a grid job, if the grid middleware at a cluster fails to invoke the job submission command because the batch server was busy and hence the command timed out, the grid middleware will interpret the job submission to be a failure and return an error to the grid user. As another example during the polling for local jobs if the batch system command to check the status of local jobs fails, because of a transient network failure, the grid middleware will again interpret the grid job to have failed. Such failures will needlessly cause the grid job to fail.

In such cases an interactive user, who sees the output of batch system commands, will simply reissue the command after a few minutes and continue working. Moreover the grid user will not be able to determine the exact cause of failure in such cases. Since almost all the batch system commands trigger some sort of network communication with a server they are particularly vulnerable to such transient failures. This problem is exacerbated when there are a number of jobs running in the batch system which is a common occurrence in a grid scenario. Such failures can be avoided by simply retrying the command in intervals spanning over a couple of minutes. Even though the grid job will still fail if the problem is particularly severe, such retrials increase the overall robustness of the system.

Typically a grid job results in the submission of multiple local jobs at a site. There is a need to create a mapping between the grid job and the local jobs in the batch system so that the grid middleware can track the progress of the grid job and determine when it has finished. This mapping can also be used to give the grid user a better indication of the progress of the grid job. For example the grid middleware can report to the user that the grid job has created *n* number of local jobs of which *x* are running, *y* have finished and *z* are queued. The way this mapping is created is totally dependent on the batch system at a particular site.

In a cluster, every worker node has a certain amount of scratch space reserved for local jobs which serves as their working area. In a cluster environment it is important that each local job runs in its own separate scratch directory at the worker nodes. This ensures mutual isolation between jobs that get scheduled to the same node simultaneously. However not all batch systems provide support for scratch management at the worker nodes where the actual computation takes place. For example some batch systems like Condor provide full fledged scratch management while other batch systems like PBS do not have scratch management support. The interactive users who are familiar with the setup of their local cluster submit jobs that have wrapper scripts around them to perform scratch management. However a grid user cannot create such wrapper scripts for a cluster as the grid user does not know about the scratch management implementation. Thus there is a need to abstract the scratch management capabilities of the batch system from the grid user. For this the grid middleware should support scratch management for grid jobs submitted to a batch system that does not provide this service.

The results of the batch system commands need to be interpreted by the grid middleware so that the middleware can determine the outcome. Typically this is done by the having grid middleware parse the output of the batch system commands. However the output produced by commands in various batch systems differs from each other. For example some batch system represent the status of a running job simply as *running* while other batch systems may call it *active*. Thus there is a need to map the batch system specific status of a local job to a set of standard statuses that the grid middleware understands.

Another problem that is prevalent in cluster computing is what the *Black Hole Effect* [5]. In a cluster, if even a single node has a configuration problem or hardware problems which results in jobs failing quickly (much faster than the execution time of the job), it reduces the turn around time at that node. This results in the batch system scheduling more and more jobs to the same node not knowing that they will fail as well. Consequently the faulty node acts like a *black hole,* eating up a lot of jobs from the batch system queue. This problem is particularly severe when a job runs for many hours and there are hundreds of such job queued up in the batch system. Consider for example a local job runs for 10 hours. There are 100 such jobs submitted to a cluster off which 10 are scheduled and started immediately. One of the nodes in the cluster results in the job failing in less than a minute. In the view of the scheduler this node is up for selection again. Depending on the size of the cluster and the user priority there, if a job is scheduled again to the same node the same cycle will be repeated. If the jobs are continuously dispatched to the same node it will result in only 9 out of the 100 jobs finishing successfully. Common examples of faults that cause the Black Hole Effect are a faulty network interface at the node resulting in files getting corrupted and DNS miss-configurations at a worker node. An interactive user can usually spot such a

problem immediately and simply resubmit jobs to the batch system asking it to avoid the faulty node. However in the case of a grid user this is not possible because the batch system is transparent to the grid user. There is a need to maintain a list of nodes that are causing problems and avoid job submission to such nodes and subsequently if such a problem is spotted, then resubmitting the job to some other node.

The submission of a grid job to a site results in the submission of one or more local jobs to the batch system at the site. The local jobs produce files such as the standard output file, standard error file, log files, and job output. In a grid environment it is necessary to ensure that the job files produced by two grid jobs do not interfere with each other to ensure mutual isolation between grid jobs. The job files created at the head node need to be transferred back to the client machine to enable the user to determine the outcome of the grid job and debug problems. So there is a need to track all the local job files created by a grid job. Finally, when a grid job finishes, it is necessary to ensure proper clean up of its job files to prevent the disk space from unnecessarily filling up.

In most batch systems the local job files are created either in a user specified location or a default location such as the *HOME* area of the user. The directory where the job files are created must be different for each grid job. There is a need to initialize a unique working directory for each job submitted through the grid to ensure mutual isolation. This further assists returning the output of the job back to the grid user and cleaning up operations at the head node.

5. Batch system abstraction

The problems identified here are common to most batch systems. These problems can be handled within the middleware. But this will lead to really complex interfaces with the batch system and adding a new batch system to a grid infrastructure will be even more complex.

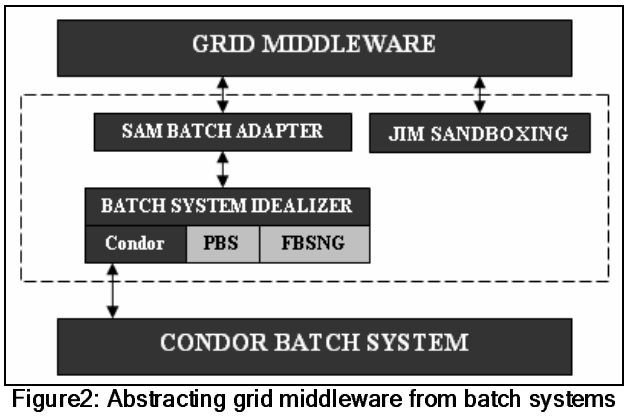

Figure2: Abstracting grid middleware from batch systems

By providing a layer of abstraction above the batch system we can shield the grid middleware from these problems, giving it the view of a *grid friendly* batch system. This layer provides the middleware with a set of services through which the middleware can interact with the underlying batch system in a uniform way irrespective of the batch system at hand. This can significantly speed up the deployment of the grid middleware. Figure 2 depicts how the grid middleware is abstracted from the underlying batch system using SAM batch adapters and the batch system idealizers. The grid middleware also uses local file management service provided by JIM sandboxing to manage grid job files as described further.

6. Batch system idealizers

Batch System Idealizers implement the interfaces required to perform batch system operations such as submitting jobs. While the batch system itself directly provides these interfaces, in order to overcome the problems noted earlier these interfaces are enhanced and are implemented in the idealizers. The idealizer scripts are totally batch system specific and to add a new batch system to the grid infrastructure, an idealizer script must be written for it.

In order to overcome the problems with transient failure in batch system commands retries are incorporated with every batch system command. The time interval for these retries is configurable, but for it to be effective it must be in the order of several minutes. This is because the typically observed failures these retrials mitigate should usually disappear in a few minutes [6]. If the problem is severe and lasts more than the retrial interval then it is best to fail and return appropriate error condition.

The Idealizers also create a mapping between the grid job and the local jobs in the batch system. To create this mapping the idealizers accept a unique identifier associated with a grid job. The batch idealizers can then associate this id with the local jobs submitted as part of the grid job submission. The way the mapping is created differs from one batch system to another. For example, in PBS the batch jobs are submitted with their *name* attribute set to the id of the grid job. In order to read the list of local jobs belonging to a grid job the PBS idealizer will search for all the jobs in the batch system queue with their name attribute set to the id of the grid job.

To provide a uniform interface of the batch system to the grid middleware the output of various commands must be uniform irrespective of the batch system. For this reason the batch idealizers convert the output of the batch system command to a uniform format. Thus the grid middleware just needs to be aware of this uniform format and not worry about different batch systems. The idealizers also perform a mapping of the batch system

status to a set of common status. The statuses that are currently supported are: *active*, *failed*, *suspended*, *pending*, and *submitted*. Thus if a batch system reports a job as *submitted* the batch idealizers will report its status as *pending* to the grid middleware.

The batch idealizers also provide scratch management support for batch systems that do not already do so. This is done by writing a scratch management script which forms the first stage of execution at the worker nodes. This script and the user executable are transferred to the worker nodes through the batch system. Upon its execution the scratch management script creates a unique directory (based on the local job id) for a job in the scratch disk at the worker nodes. The location of scratch disk at the worker nodes is read from configuration at the head node. The scratch management script then launches the user executable from under the unique scratch area for the job. When the user executable finishes, the scratch management script then cleans up the job area in the scratch disk. A problem with this scheme is that if the job is deleted from the batch system, its scratch area is left *dangling* i.e. its job area won't be cleaned up. The clean up operations of the scratch management script will not be invoked in this case. This problem may be eliminated by having the scratch management script at the beginning of its execution examine the scratch area and cleaning up any directories belonging to jobs that are no longer in the batch system queue. Thus if the scratch directory for a job is left *dangling* it will be cleaned when the next job is scheduled at that node.

Earlier, we described the Black Hole Effect problem with clusters. While solving this problem in an automated way is complex, the batch idealizers may alleviate its effect by maintaining a *neglect list*, which contain the names of the nodes discovered to have problems. During job submission the idealizers explicitly ask the batch system not to schedule jobs to nodes in the neglect list. Currently this list is being maintained manually and whenever a problem is identified the site administrator will need to update this list. The manual interference of the administrator does not solve the problem for the grid user. However once the computation of the neglect list is automated, the grid user can resubmit jobs knowing that it won't suffer the same problem again.

7. SAM batch adapters

SAM batch adapter [7] is a package developed at Fermilab [8] as part of the SAM project [9]. We have adopted this package as a configuration tool that provides the grid middleware with interfaces to invoke the appropriate batch idealizer at a site. While the idealizers implement the interfaces to allow interactions with the batch system, the grid middleware still needs to know how to invoke them. This is accomplished through SAM batch adapters. Thus the batch system idealizers combined with SAM batch adapters provide a complete abstraction of the underlying batch system to the grid middleware. SAM batch adapter package has many features; here we just discuss the aspects of the package that are relevant within. For a more detailed reading on the topic refer to [7].

SAM batch adapter contains in its configuration the batch idealizer commands that implement various batch system operations. The configuration of the package is stored in a local Python module which can be updated using an administrative interface the package provides. Figure 3 shows a part of SAM batch adapter configuration. Each command stored in the configuration has a *command type* associated with it. The command types that we use are – *job submit command*, *job kill command*, and *job lookup command*. The function of a command can be derived from their types.

Each command has a *command string* associated with it which may contain any number of predefined *string templates*. String templates are used for plugging the user input into a command string, which then gives a command that the user or API client can execute to get the desired results. For example in figure 3 the command string for the job lookup command is "…/sam_condor_handler.sh job_lookup --project=%__USER_PRO JECT__ --local-job-id=%__BATCH_JOB_ID__". In order to perform lookup operation the API client or the user can read the command string giving its command type (in this case "job lookup command") and then replace the template strings with user input. The template strings in this case are "%__USER_PROJECT__" which needs to be replaced with the grid id of a job and "%__BATCH_JOB_ID__" which optionally needs to be replaced with a local job id if performing lookup on a single batch job. The resulting command string when executed will invoke the batch idealizer's (in this case a Condor idealizer) lookup operations based on the grid id.

Each batch command can have multiple results or possible outcomes associated with it. The result is characterized by the exit status of the command and may have an output string associated with it which may contain a string template. The exit status in question here is the status which is returned by the operating system when the command is executed after template substitution. In figure 3 there are three results associated with the job lookup command. The first result says that an exit status 0 corresponds to success. The second result extends this by saying that the output produced by the command upon its successful execution is a list batch job ids and their status. The third result states that an exit status of 1 means that the command has failed.

```
batchCommandResult_1 = BatchCommand.BatchCommandResult(0, "", "Success")
batchCommandResult_1 = BatchCommand.BatchCommandResult(0,
     "JobId=%__BATCH_JOB_ID__ Status=%__BATCH_JOB_STATUS__", "Success")

batchCommandResult_2 = BatchCommand.BatchCommandResult(1, "", "Failure")
batchCommand_4 = BatchCommand.BatchCommand("job lookup command",
     "${SAM_BATCH_ADAPTER_HANDLER_DIR}/sam_condor_handler.sh job_lookup
     --project=%__USER_PROJECT__ --local-job-id=%__BATCH_JOB_ID__",
     [batchCommandResult_1, batchCommandResult_2, ])
```

Figure 3: SAM Batch Adapter Configuration

It is worthwhile to note that the SAM batch adapter itself does not execute the commands to perform batch system operations. It just provides a functionality to prepare commands for execution. It is the responsibility of the API client to execute commands and interpret their results. The commands that get executed are the batch idealizers with their enhancements to the batch system interfaces.

As mentioned earlier there are many string templates defined in SAM batch adapter, but only a few are used in our scheme. Table 2 lists the string templates used in our scheme along with their purpose.

Table 2: Use of string templates

Template String	Purpose
%__USER_PROJECT__	Specify the grid id of a job to the idealizer scripts
%__USER_SCRIPT__	Specify the name of the executable to be submitted to the local batch system
%__USER_SCRIPT_ARGS__	Specify the arguments if any, to the executable submitted to the local batch system
%__USER_JOB_OUTPUT__	Specify the path where the standard output file of the batch job should be deposited
%__USER_JOB_ERROR__	Specify the path where the standard error file of the batch job should be deposited
%__BATCH_JOB_ID__	Specify the id of a single batch job
%__BATCH_JOB_STATUS__	Specify the current status of a batch job (used mainly in command results)

8. JIM sandboxing

JIM Sandboxing [10] provides a local file management service to the grid middleware. It is a tool used to initialize the relevant input files for a job and return a collection of all the output and diagnostic files produced by a grid job. Normally, when a job is submitted interactively to a batch system, the standard output and error files are deposited in either a user specified location or a default location such as the home area of the user. In a grid environment the user cannot provide this information, it is transparent to the user. This can be set to some fixed location configured at each site or some other default location. However it will result in multiple grid jobs that are running in parallel producing there job files under the same path at the head node. In this case keeping track of the job files of a grid job becomes difficult as a typical job will have hundreds of job files associated with it. If two grid jobs produce a file with same name it will interfere with their execution violating their isolation. This also complicates the collection of job files for a grid job which need to be transferred back to the client machine and the cleanup of the job files.

JIM Sandboxing provides the mutual isolation between two grid jobs by initializing a unique *sandbox area* for each grid job. A sandbox is a directory on a local disk which is serves as the working area for the grid job. The grid middleware can instruct the batch system to create the standard output and error files for a batch job in its sandbox area. All the job files produced by local jobs belonging to a grid job are deposited in the sandbox area for that grid job.

The sandbox area also serves as a staging area for the input files needed by the batch jobs. JIM Sandboxing supports the concept of an *input sandbox* which is a collection of user supplied input files needed for the execution of local jobs. The user can supply the input sandbox at the time of grid job submission and it can be transferred to the head node through the grid middleware and unpacked in the sandbox area of the grid job.

Once the grid middleware has initialized the sandbox area for a grid job it can then *package* it. During the packaging of a sandbox, a control script is created which forms the executable that is submitted to the batch system. When launched, this control script copies all the contents of the sandbox area from the head node to the worker nodes and launches the user executable. JIM Sandbox also provides an interface to collect all the job files or output files present in the sandbox area of a grid job. Using this grid middleware can easily transfer the output of a grid job back to user machine.

9. Integration with Condor-G and Globus

The tools and methods described here have been put to use in the SAM-Grid project [11] based at Fermilab. The grid middleware used in SAM-Grid is Condor-G system [12] which combines software from Condor with Globus. In Condor-G there is a process called the *gatekeeper* running on the head node of a cluster that can be invoked by the grid scheduler. The gatekeeper executes a process called the job manager for each grid job submitted to the cluster. Here we discuss how the job managers in SAM-Grid make use of the tools described, to interact with the batch system.

When a job is submitted, the job managers initialize a unique working area for the grid job using the JIM Sandbox interface. If there is any input sandbox transferred by Condor-G then it is unpacked into the sandbox area and then the job managers package the

sandbox area. Using SAM batch adapter the job managers read the command string for the job submit command. Then template substitutions are performed replacing the executable template with the sandbox control script, the standard output and error file templates with the path to the sandbox area and filenames and finally the grid id template with the id of the grid job. Then the resulting command is executed to submit jobs to the batch system. In order to submit multiple local jobs the job managers simply need to execute the same command multiple times.

For checking the status of the grid job the job managers need to read the job lookup command through the batch adapters and then perform template substitution appropriately and execute the resulting command. The job managers can parse the output of job lookup commands and determine the status of the grid job.

The job manager operations described above are same at all the sites irrespective of the batch system being used there. Thus this enables writing a uniform job manager that can be deployed at all sites. The batch systems that have been incorporated into the SAM-Grid i.e. the batch systems for which an idealizer has been implemented are – The batch at CC-IN2P3 (BQS), the Portable Batch System, the Condor Batch System and Farms Batch System Next Generation.

The SAM-Grid project is in use for running physics applications such as Monte Carlo simulations. Here we quote some figures about the performance of the grid infrastructure from the SAM-Grid project. Over a period of 9 months (Jan 2004 through Sep 2004) SAM-Grid has delivered 17 years worth of computation on a 1 GHz computer [13]. The overall efficiency of the grid infrastructure that has been measured over this interval is close to 99%.

10. Conclusions

The grid computing technologies in use today are not completely isolated from the batch system that is being run on a cluster. By providing a layer of abstraction between the batch systems and the grid middleware we have bridged the gap between the local job management provided by various batch systems and the grid level job management provided by the grid middleware. It has resulted in a system that can be easily incorporated with any grid middleware for easily connecting clusters to a grid.

A new batch system can be incorporated into a grid by simply writing an idealizer script for the batch system. This does not require any knowledge about the functioning of the middleware. In addition to this standard software can be distributed with the grid middleware which interfaces with the tools described here and so the grid middleware need not deal with different batch systems. This significantly speeds up the deployment of a grid middleware. The idealizers also mitigate the deficiencies in batch systems, which we identified in section 4 that makes their integration into a grid difficult increasing the overall robustness of the system. The sandboxing mechanism allows for easy file management of grid jobs and also ensures mutual isolation between grid jobs.

11. References

[1] The Globus Alliance home page, www.globus.org

[2] Open PBS home page, http://www.openpbs.org

[3] Tannenbaum, T., et al., "Condor - A Distributed Job Scheduler", in Thomas Sterling, editor, *Beowulf Cluster* The MIT Press, 2002. ISBN: 0-262-69274-0

[4] Farm Batch System Next Generation, http://www.isd.fnal.gov/fbsng

[5] Nishandar, A., et al., "Black Hole Effect: Detection and mitigation of application failures due to incompatible execution environment in computational grids", submitted to CCGRID-2005

[6] The SAM-Grid deployment issues, http://www-d0.fnal.gov/computing/grid/deployment- issues.html

[7] SAM Batch Adapters, http://d0db.fnal.gov/sam _batch_adapter/ sam_batch_adapter.html

[8] Fermi National Accelerator Laboratory, www.fnal.gov

[9] Carpenter, L., et al., "SAM Overview and Operation at the D0 Experiment" in the proceedings of Computing in High Energy and Nuclear Physics, Beijing, China, September 2001

[10] Garzoglio, G., et al., "The SAM-Grid Fabric Services" in IX International Workshop on Advanced Computing and Analysis Techniques in Physics Research (ACAT), Tsukuba, Japan 2003

[11] Garzoglio, G., et al., "SAM-GRID project: architecture and plan" at the 8th International Workshop on Advanced Computing and Analysis Techniques in Physics Research (ACAT-02), Moscow, Russia, Jun. 2002, Published in Nuclear Instruments and Methods in Physics Research, Section A, NIMA14225, vol. 502/2-3 pp 423 – 425

[12] Frey, J., et al., "Condor-G: A Computation Management Agent for Multi-Institutional Grids" in the proceedings of Tenth IEEE Symposium on High Performance Distributed Computing, San Francisco, California, August 2001

[13] Garzoglio, G., et al., "Experience producing simulated events for the DZero experiment on the SAM-Grid" in the proceedings of Computing in High Energy and Nuclear Physics, Interlaken, Switzerland, September 2004