ABSTRACTING THE HETEREOGENEITIES OF COMPUTATIONAL RESOURCES IN THE SAM-GRID TO ENABLE EXECUTION OF HIGH ENERGY PHYSICS APPLICATIONS

The members of the Committee approve the master's
thesis of Sankalp Jain

Supervising Professor Name
David Levine ____________________________________

Jaehoon Yu ____________________________________

Leonidas Fegaras ____________________________________

ABSTRACTING THE HETEREOGENEITIES OF COMPUTATIONAL RESOURCES IN THE SAM-GRID TO ENABLE EXECUTION OF HIGH ENERGY PHYSICS APPLICATIONS

by

SANKALP JAIN

Presented to the Faculty of the Graduate School of

The University of Texas at Arlington in Partial Fulfillment

of the Requirements

for the Degree of

MASTER OF SCIENCE IN COMPUTER SCIENCE AND ENGINEERING

THE UNIVERSITY OF TEXAS AT ARLINGTON

December 2004

ACKNOWLEDGEMENTS

I would like thank my supervising professor, Mr. David Levine for his constant encouragement and guidance throughout my research.

I am grateful to Dr. Jaehoon Yu, Department of Physics, UT Arlington for his support and encouragement throughout my research and for managing the UTA-FNAL exchange program.

I would like to thank Dr. Leonidas Fegaras, Department of Computer Science and Engineering, UT Arlington for serving on my committee.

I would like to thank the core SAM team at Fermilab, specifically Andrew Baranovski, for their help during my work

Most importantly, I would like to express my sincere gratitude to the SAM-Grid team at Fermilab, specifically Igor Terekhov and Gabriele Garzoglio. Without their support and guidance I would not have been able to complete the work in this thesis. I would also like to thank Aditya Nishandar for his help during my work.

I would like to thank Abhishek Rana, Siddharth Patil, Hannu Koutaniemi, Parag Mhashilkar and Vijay Murthi. Although I never worked with them, their work prior to my arrival at Fermilab proved to be invaluable for the work in this thesis.

I am thankful to my parents and sister for their constant support and encouragement throughout my academic career.

November 10, 2004

ABSTRACT

ABSTRACTING THE HETEREOGENEITIES OF COMPUTATIONAL RESOURCES IN THE SAM-GRID TO ENABLE EXECUTION OF HIGH ENERGY PHYSICS APPLICATIONS

Publication No. ______

Sankalp Jain, M.S.
The University of Texas at Arlington, 2004

Supervising Professor: David Levine

The computing infrastructure of modern High Energy Physics collaborations consists of resources that are spread across the globe and that are greatly different from one another – different batch systems, file systems, cluster configurations, etc. A Grid that unites all these resources must make these differences transparent to the user and the application being run. The work in this thesis has been done as part of the SAM-Grid project at Fermi National Accelerator Laboratory, Batavia, IL and has resulted in a system that abstracts the user application from the heterogeneities of a Grid providing user applications with a uniform working environment and one that is easily deployable and maintainable. We have used Grid middleware tools such as the Globus toolkit and other tools developed at Fermilab to create a uniform working environment at the worker node where the application is actually executed. This system has been deployed on 12 sites in 6 countries and is currently being used for running High Energy Physics applications for the D0 and CDF experiments based at Fermilab.

TABLE OF CONTENTS

LIST OF ILLUSTRATIONS

CHAPTER 1

INTRODUCTION

In recent years Grid Computing has emerged as a popular and useful paradigm for solving large computational and data intensive problems. The evolution of Grid Computing has been driven by the increasing complexities of the problems that the scientists in today's age are trying to solve. Problems such as molecular drug design or simulation of High Energy Physics particle collisions require an unprecedented amount of computational and storage resources.

1.1 What is Grid Computing?

Grid Computing is a distributed computing paradigm which makes it possible for scientific collaborations and other institutions to share their resources at an unprecedented scale and for geographically distributed groups to work together in ways that were previously impossible [FOSTERPHYSICSTODAY]. A Grid is a system that is built on the Internet and provides a scalable, secure, & high-performance mechanism to coordinate and share resources across dynamic and geographically dispersed organizations. This sharing involves direct access to computers, software, data, and other resources. A Grid can seamlessly integrate the resources (often geographically distributed) of organizations or scientific collaborations, allowing for better resource management by providing transparent access to the resources of the collaboration to all members.

1.2 Grid Computing and other distributed computing paradigms

The problem that differentiates Grid Computing from other distributed computing paradigms such as Enterprise distributed computing technologies is the distributed ownership of the resources and the unique resource sharing requirements. In a Grid environment the resources are under different ownership or administration. The *resource consumer* (users of the resource) may be from an organization other than the *resource provider* (owner of the resource). Hence the sharing of resources is highly controlled with resource providers and consumers defining clearly just what is shared, who is allowed to share, and the condition under which the sharing occurs [ANATOMY]. A group of individuals or institutions defined by such sharing rules is termed a *Virtual Organization* (VO). In other words a VO is a collection of individuals and institutions that pool their resources together to achieve a common goal. [ANATOMY] lists the following as an example of a VO: an application service provider, a storage service provider, a cycle provider, and consultants engaged by a car manufacturer to perform scenario evaluation during planning for a new factory. Another example of a VO is a modern High Energy Physics collaboration in which geographically distributed institutions pool their resources together to solve common problems.

[ANATOMY] identifies the following as the main requirements and concerns of the Grid domain that differentiates it from other distributed computing paradigms:

1. Highly flexible sharing relationships for precise control over how resources are used. The sharing relationships vary from client-server to peer-to-peer.
2. Fine-grained and multi-stakeholder access control, delegation and application of local and global security policies. The access to the resources needs to be governed by the policies set by the owners (local policies) i.e. there cannot be a single access control policy for all the resources of a VO.
3. The resources themselves vary from programs, files, and data to computers, sensors, and networks.
4. Diverse usage modes ranging from single user to multi-users and from performance sensitive to cost sensitive hence embracing the issues of quality of service, scheduling, co-allocation, and accounting.

These requirements are not solved by the distributed computing technologies in use today. For example, current internet technologies address communication and information exchange among computers but do not provide integrated approaches to coordinated use of resources at multiple sites [ANATOMY]. Enterprise distributed computing technologies such as CORBA and Enterprise JAVA enable resource sharing within a single organization [ANATOMY]. They either do not accommodate the range of resource types or do not provide the flexibility and control on sharing relationships needed to establish VOs.

1.3 High Energy Physics at Fermilab

Fermi National Accelerator Laboratory (Fermilab) [FNAL] located at Batavia, IL is home to the world's highest energy particle accelerator, the Tevatron. The Tevatron accelerates protons and anti-protons close to the speed of light and then makes them collide head-on [CDF-1]. These collisions are recorded by detectors inside the Tevatron and are converted it to digital data which is then stored to permanent storage. The physicists study the products of such collisions and try to reconstruct what happened in the collision and ultimately try to figure out how matter is put together and what forces nature uses to create the world around us [CDF-1].

Currently there are two main experiments based at Fermilab that are making use of the Tevatron. The DZERO experiment is a collaboration of over 500 scientists and engineers from 60 institutions in 15 countries [D0-1]. The other experiment, Collider Detector at Fermilab (CDF) is also a collaboration of physicists and institutions from around the world [CDF-1]. These experiments are in data taking phase referred to as *RUN-II* and each experiment produces data in the order of TB/day. The previous phase called *RUN-I* lasted through the beginning of 1996 and resulted in important discoveries by both the experiments that have helped scientists to better understand the nature of matter.

1.4 High Energy Physics and Grid Computing

As described in the last section the High Energy Physics (HEP) experiments generate data in the order of TB/day. This data is stored on a Mass Storage System based on tape archives called Enstore developed at Fermilab [ENSTORE]. The Enstore provides a generic interface (an API) so experimenters can use Mass Storage System as easily as if they were native file systems. Analyzing the data stored in the Mass Storage System requires a large amount of computational resources. The computational resources of the experiments are distributed across various institutions around the world. Thus there is a need to provide distributed access to the data and also to the resources of the experiments so that the experimenters can study the data irrespective of their geographic location.

The HEP experiments resemble the Virtual Organization concept described in section 1.2. In particular the resources of a HEP experiment are owned by different institutions that are a part of the experiment and there is a need to allow access to these resources to any individual who is a part of the experiment. Thus Grid Computing presents itself as an ideal solution for the computational requirements of HEP experiments such as DZERO and CDF. To provide distributed access to the data in the Mass Storage System, a system called SAM (Sequential Access to Data via Metadata) has been developed internally at Fermilab [SAM-1]. The SAM system is described in detail in the next chapter. The SAM-Grid project at Fermilab is focused towards providing Grid solution to the HEP experiments based at Fermilab. The goal of the project is to create a Grid infrastructure that will enable scientists to access the resources belonging to the experiment as if they were local resources. The SAM-Grid project is funded by the Particle Physics Data Grid [PPDG].

1.5 Goal of the Thesis

The work in this thesis has been done as part of the SAM-Grid project at Fermilab and it has been supported by the Department of Physics, the Department of Computer Science and Engineering at the University of Texas at Arlington, Arlington, TX and the Fermi National Accelerator Laboratory, Batavia, IL.

The goal of the thesis is to develop new methods and systems that abstract the differences that are inherent in the computational resources of a Virtual Organization and at the same time eliminating the fallacies in such resources which may make their integration into a grid difficult. This is done providing a level of abstraction between the resource managers of the computational resources and the grid middleware.

The rest of the thesis is organized as follows: Chapter 2 provides an overview of the various technologies used in SAM-Grid. Chapter 3 discusses in detail the architecture of SAM-Grid. Chapter 4 describes the methods and system developed to allow integration of diverse resources into SAM-Grid. Finally chapter 5 lists the Conclusions that can be drawn from this thesis.

CHAPTER 2

RELATED WORK

The implementation of the SAM-Grid system makes use of many Grid computing technologies. This chapter provides an overview of the Grid technologies used in SAM-Grid. It also presents an overview of SAM, the data handling system that SAM-Grid depends on.

2.1 Globus

The Globus Alliance is developing fundamental technologies needed to build computational grids [GLOBUS-PAGE]. The Globus Toolkit is an open source software toolkit used for building grids. It is being developed by the Globus Alliance and many others all over the world [GLOBUS-TOOLKIT]. The Globus Toolkit includes software services and libraries for resource monitoring, discovery, and management plus security and file management. The rest of this section describes the tools in Globus Toolkit used in SAM-Grid.

2.1.1 GSI (Grid Security Infrastructure)

The Grid Security Infrastructure is a part of the Globus Toolkit that provides fundamental security services that are needed to build grids [GSI-1]. The GSI implementation is based on public key infrastructure, X.509 certificates and SSL/TSL protocol.

X.509 certificates alone are not good enough to meet the security requirements in a grid environment for reasons described further. It is often the case that a grid user needs to delegate a subset of their privileges to another entity on relatively short notice for a brief amount of time [GSI-1]. X.509 certificates do not support this form of delegation. Also the private keys associated with X.509 certificates are often protected by pass phrase or pins. This prohibits repeated authentication without user intervention. This concept is known as *single sign-on*. The GSI extends X.509 certificates and allows for dynamic delegation of credentials using the SSL protocol and single sign-on by creating a new set of credentials that do not require any pass phrases or pins from the user and are valid for a small amount of time.

2.1.2 GRAM (Globus Resource Allocation Manager)

Globus Resource Allocation Manager is an interface by which remote system resources can be requested and used [GRAM-URL]. It is a set of components that are bundled as part of the Globus Toolkit. GRAM is one of the most widely used interfaces for remote job submission and resource management in the grid community.

Under GRAM there is a process called the *gatekeeper* running on the machine hosting the remote resource. This machine is called the *gateway or the head node* at a remote site. The request for access to a resource is composed in Resource Specification Language (RSL) which the gatekeeper understands. When the gatekeeper receives such a request it uses GSI to authenticate and authorize the user. For this the gatekeeper refers to a file called the *grid-map file* which is a mapping of the users' GSI credentials to a local account on the host machine. Thus the users' privileges at the remote site are limited to that account's privileges at the site. This feature provides the flexibility of having the local security policies take precedence over global policies. Thus the access to a resource is totally governed by the administrators of various resources, an important requirement in grid environment. Usually there is one local account for each Virtual Organization that the resource is participating in.

Once authorized the gatekeeper passes the request to a *job manager* which carries out the requested operation. The GRAM protocol has also the provision to stage files from the client machine using Global Access to Secondary Storage [GASS]. In the most common scenario the gatekeeper is running on the submit node (the node from where jobs can be submitted to the batch system) or the head node of a cluster and the job manager simply submits job to a local batch system.

2.1.3 GridFTP

GridFTP is a high-performance, secure, reliable data transfer protocol optimized for high-bandwidth wide-area networks [GRIDFTP-URL]. It is based on the File Transfer Protocol (FTP), the highly-popular file transfer mechanism widely deployed on the Internet.

[GRIDFTP-DOC] lists the following GridFTP extensions to FTP

- GSI security on data and control channels: Robust and flexible authentication and integrity features are critical when transferring or accessing files.
- Multiple data channels for parallel transfers: On wide-area links, using multiple TCP streams can improve aggregate bandwidth over using a single TCP stream.
- Third party (direct server-to-server) transfers: In order to manage large data sets for large distributed communities, it is necessary to provide third-party control of transfers between storage servers.
- Striped data transfer: Partitioning data across multiple servers can further improve aggregate bandwidth.

GridFTP has become very popular in grid community because of its support for GSI and the scalability of FTP.

2.2 Condor-G

Condor is a workload management system for high throughput computing. Condor can be run in an *idle cycle hunting* mode where idle cycles from computers that are currently idle can be harnessed to do some useful processing. Like other full featured batch systems, Condor provides a job queuing mechanism, scheduling policy, priority scheme, resource monitoring, and resource management [CONDOR-URL]. Amongst the novel feature of Condor are the Condor ClassAds and Check Pointing. The ClassAd mechanism in Condor provides a very flexible and expressive way of matching resources requirements (jobs) with resource offers (machines) [CLASSADS]. A ClassAd for a job contains the requirements of the job such as required operating system, memory needed etc. A resource ClassAd contains information about the resource capability. The scheduling a job is done by matching the job ClassAd against those of the resource and finding a resource capable of satisfying the job requirements. The Condor checkpointing mechanism allows the state of a job running at a particular machine to be saved and the job can be continued at another machine. This allows for migration of jobs form one host to another without loss of computational time. This allows Condor to be run in an *idle cycle hunting* mode in which jobs are scheduled as resources become available (when the resource is not claimed by the owner) and they can be evicted once the owner reclaims the resource. The Condor system is the product of Condor Project at the University of Wisconsin at Madison.

Condor-G system leverages software from Condor and Globus to allow users to harness multi-domain resources as if they all belong to one personal domain [CONDORG]. Condor-G combines the resource brokering capability of the Condor system with GSI and GRAM from the Globus Toolkit resulting in a grid like system. In Condor-G the resource brokering is provided by the Condor Match Making Service in which the ClassAds from the job are matched against the ClassAds from the remote resources to find a best match for the job. Once a resource has been identified Condor-G then submits the job to the resource using GRAM and GSI from the globus toolkit. The details about the functioning of Condor-G within the SAM-Grid system are described in detail in the next chapter.

2.3 SAM (Sequential Access to data via Metadata)

SAM is a data handling system developed at Fermilab to handle high volumes of data (in the order of TB/day). SAM stands for *Sequential Access to data via Metadata* where sequential refers to sequential physics events stored within files which in-turn are stored sequentially on tapes within a Mass Storage System (MSS) at Fermilab [SAM-1]. It was originally developed for the DZERO experiment based at Fermilab. SAM has distributed network architecture as shown in figure 2.1 which is taken from [SAM-1].

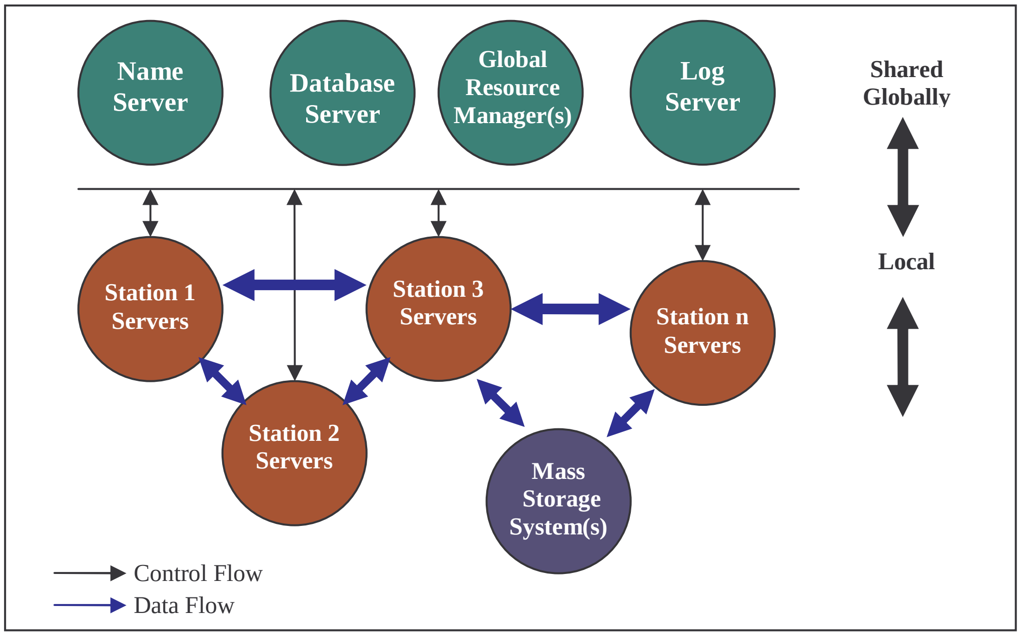


Figure 2.1: Overview of SAM

The system can be divided into a set of shared or global components and local or site specific components. The different components of the system communicate with each other using CORBA [SAM-1]. In SAM the catalogs of metadata, locations (replica catalogs), data transformations, SAM configuration and policy, detector configurations and other parameters are maintained in a central Oracle database at Fermilab [SAM-2]. There are more than one database servers servicing requests to this database. In current infrastructure there is a database server to service the requests coming from SAM stations, a server for interactive user requests, a server for requests coming as a result of grid activity, and a server for Web queries. This paradigm provides scalability and as more local components are added to the system, if need be the number of database servers can be increased proportionally. The actual physics data coming out of the detectors and the simulated & reconstructed data is stored in the MSS like the Enstore at Fermilab. The different remote processing centers can access the data in Enstore using the Metadata in the central repository.

The following is the list of globally shared services in SAM as shown in Figure 2.1.

- A CORBA naming service that allows all other components in the system to find each other by name
- The database servers providing access to the Metadata database
- One or more global resource managers deployed to control and more effectively manage resources like Automated Tape Libraries.
- A log server that is used to gather logging information from the entire system.

These global services are accessible to the local components at various sites in the system. All the network communication between different components uses well defined CORBA interfaces.

In SAM each local processing site or center is called a SAM Station. Figure 2.2 taken from [SAM-1] shows the internals of a SAM Station. Each station has a logical disk called *Station Cache* associated with it. The process responsible for managing this cache is called the *Station Master*.

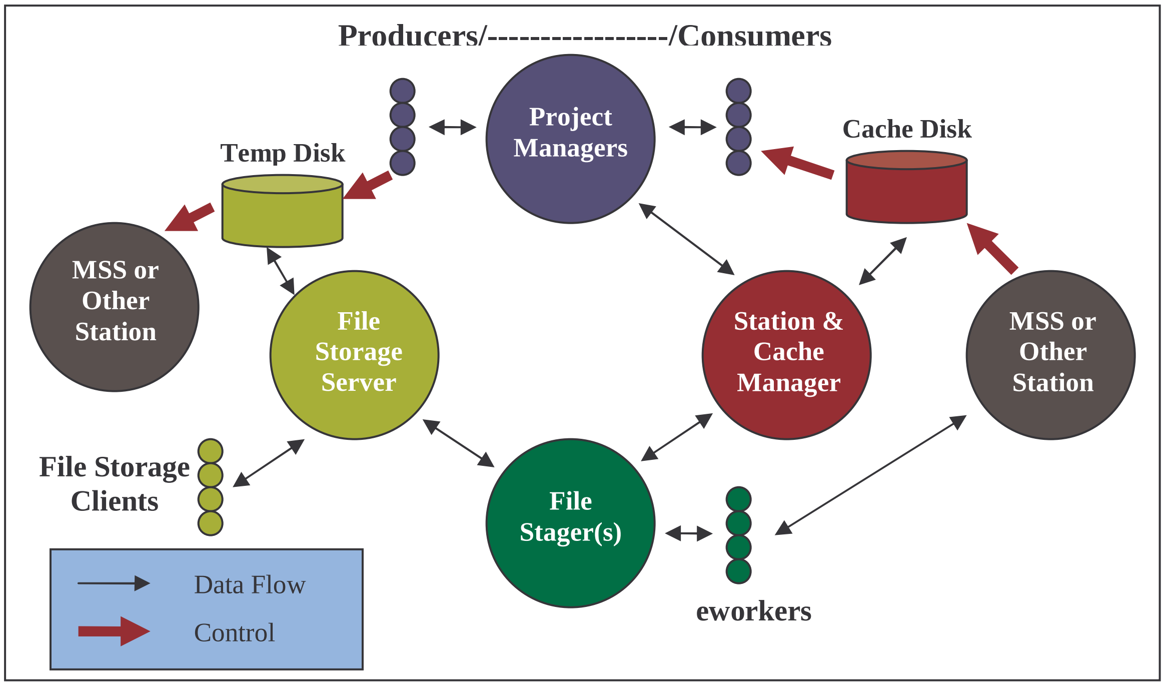


Figure 2.2: SAM Station Components

Different SAM stations can communicate with each other using *Station Names* and the CORBA naming service. Thus data in one station's cache can be replicated in another stations cache on demand, with the central replica catalog maintaining the valid list of locations. Thus if a particular file is present in some station's cache then it can be directly copied to the requesting station's cache rather than getting it from MSS which has a longer delay associated with it.

The part of the station that is responsible for moving data from one Station to another is called the *File Stager*. SAM supports a wide variety of protocols for transferring files – GridFTP, rcp, kerberized rcp, ftp, encp (for copying files from Enstore) etc. To move a file from one station to another the receiving station's master instructs its stager, which then invokes a process called *Eworker* for each file transfer. The Eworkers perform the actual copy operations. The stager keeps track of each file transfer and retries each failed transfer a configurable number of times. Storing of files to the MSS is the responsibility of the *File Storage Server*. For storing a file first its metadata is declared or stored in the central repository and then the File Storage Server performs the actual file transfer to the MSS.

In order to access files stored in the SAM system a user must first define a *Dataset*, which is a set of files matching certain criterion. There is a command line interface, a web interface, and a Python API available to define datasets of files available in SAM. In order to fetch a dataset the user must run a *SAM Project* against the dataset. *Project* is the name given to the process of getting files from the SAM system. When a SAM project is submitted to the station master it launches a child process called the *Project Manager* for each project, which is responsible for getting the files in the dataset from, either MSS or other stations, to the station's cache. User applications consume the files by a process called *the Consumer* which registers itself with the station master. The files are pulled by the consumer in the sense that when a consumer asks for the next file, if it is available in the local cache the Project Manager returns its location to the consumer which can then copy it to user's area. If a file is not available the consumer blocks until it is signaled by the Project Manager. There are command line interfaces and also a python API available for running and monitoring SAM Projects. Using these commands the users can determine the status of a project which includes the number of files acquired in station cache, the number of files consumed by the user application, the number of files in the dataset against which the project is being run etc.

As the stations are governed by local security policies, not all stations can directly communicate with each other (for example due to firewall issues). Also only a certain stations have direct access to the MSS. To solve this problem SAM supports the concept of *file routing*. If a file lies in a part of the system that is not directly accessible to a station then the station's access to the file is defined via route through one or more SAM stations that provide caching and forwarding services [SAM-2]. There is an accessibility matrix for each compute and storage element managed or accessed by a station.

2.4 SAM-Grid Project

The SAM-Grid project at Fermilab is an effort to build a grid that uses SAM as its data handling system to enable fully distributed computing for the DZERO and the

CDF experiments. SAM is a robust and scalable data handling system but it lacks grid level job submission and management features. This has been accomplished by the development of Job and Information Management (JIM) system. JIM has been developed using the grid technologies described in this chapter – Condor-G and Globus. The SAM-Grid project is funded by PPDG and GridPP and has been in use for the last year to run Monte Carlo simulations for the DZERO experiment and is in the process of being adapted for the CDF experiment.

CHAPTER 3

SAM-GRID ARCHITECTURE

The SAM-Grid architecture can be organized into two layers: the *Grid Layer* which encompasses the services that are global in nature, and the *Fabric Layer* which includes services whose scope are restricted to the scope of a or computational resource [CHEP04]. Fabric is the collection of all the physical computational and storage resources that are part of the grid. The grid layer is on top of the Fabric layer and is responsible for providing the connectivity amongst the resources of the grid. The two layers are responsible for providing different set of services that together complement each other to provide a working grid system. This chapter describes the two layers, the services they offer and the interface between the two layers.

3.1 The Grid Layer in SAM-Grid

The grid layer in the SAM-Grid architecture provides the connectivity amongst the distributed resources of a Virtual Organization. The word connectivity here implies a *grid level job management* that the grid layer provides to enable job submissions to remote sites (computational resources). These computational resources are cluster of computers that are managed by a local batch system at each site. The grid level job management is different from *local job management* which is provided by the local batch system managing the resource. The rest of this section discusses the issues involved in the grid level job management and its implementation in the SAM-Grid system.

When a user submits a job to run on a grid, he/she specifies the executable to run, its input if any and the requirements of the resource where the job can execute i.e. the job requirements. Once a job is submitted it is the responsibility of the grid layer to find a resource on the grid that matches the job requirements. Once such a resource has been identified the grid layer must start the execution there or if no such resource exists return an error to the user. In order to provide this grid level job management the grid layer must provide the following services.

- *Resource Discovery*: A grid has a number of computational resources connected to it. The grid layer must know about all the resources that are a part of the grid and their capabilities such as the amount of physical memory available, the processing capabilities or the size of the resource etc.
- *Resource Matching / Brokering*: The grid layer has to match the user supplied job requirements with the requirements of all the resources in order to find a resource that is best capable of running the user executable.
- *Job Submission*: Once a resource capable of running a job has been identified, the grid layer needs to submit the job to that resource. This involves authenticating the user, staging in the user executable to the remote site along with its input if any and staging out the output back to user machine.

- *Job Monitoring*: The grid layer needs to keep track of the progress of a job, determine when it has finished, and report the current status of active jobs. Also it must also provide a way for the user to terminate or kill a *grid job*.

The grid layer needs to provide the services described above in the context of a Virtual Organization. For example, if the user belongs to a certain Virtual Organization then the grid layer must deal with only the resources that belong to that Virtual Organization.

The Condor-G system described in the last chapter provides the services described above. The SAM-Grid architecture has adopted the Condor-G system as the core of its grid Layer. However the SAM-Grid architecture depends on SAM rather than Condor-G, to provide the data handling needed by the system such as the staging-in of the user executable and its input, staging out of the output etc. This is because the applications being run on SAM-Grid are all dependent on SAM for getting their input and storing their output. The application executables themselves are very large in size for example the size of one release of the DZERO Monte-Carlo application is about ½ gigabytes. So it makes sense to use the rich caching features of SAM at the remote site rather than transferring the executable to the execution site, every time a job is submitted there. The Condor-G stage-in is used only for transferring some small user files such as user specific configuration files. Also the Condor-G stage-out is used for carrying the standard output and error streams & other log files produced by the execution of a job back to the user machine.

As shown in figure 3.5, the grid layer in SAM-Grid consists of Condor-G system, the global data handling services of SAM and a Web server to enable monitoring via the World Wide Web.

3.1.1 The Grid Layer Design

The grid layer in SAM-Grid has a *3-Tiers architecture* in which distinction is made between the *submission site* where the jobs are submitted and scheduled, the *broker site* where the resource brokering is done, and the *execution site* where computational resources are hosted as illustrated in figure 3.1. These three sites are conceptual entities and they may either reside on the same machine or on different machines, the latter being the most common case. There can be any number of each site in the system i.e. multiple submission sites, multiple execution sites and/or multiple brokering sites. The current infrastructure has just one brokering site, 7 submission, and 12 execution sites. This architecture has proved to be very flexible, scalable, and robust. Figure 3.1 illustrates the control flow in the grid layer resulting from a single job submission. For simplicity the global services of SAM are not shown in the figure.

The resource discovery service is provided using the Condor ClassAd mechanism [CLASSADS]. At each execution site a process called C*ondor Advertise* periodically sends a *ClassAd* to the resource broker. This ClassAd contains information about the execution site like the URL of the gatekeeper, the amount of physical memory available etc. At the resource broker a process called *Condor Collector* collects or receives the ClassAds from all the execution sites. This enables the grid layer to gather information about all the execution sites.

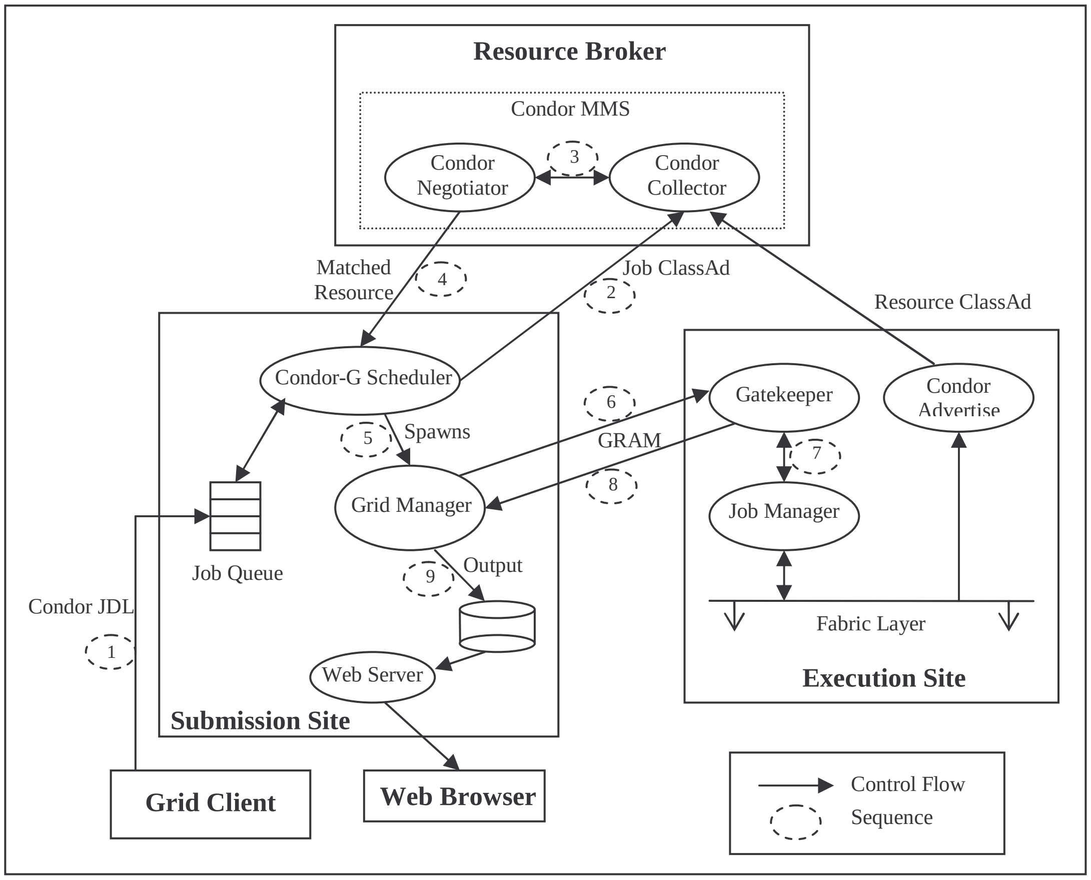
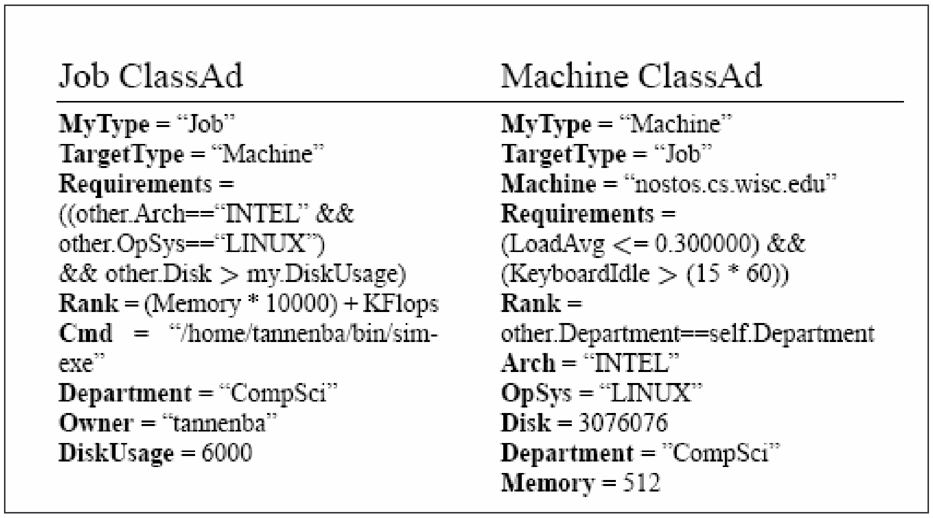


Figure 3.1: Grid Layer Control Flow for a single job

At the submission site where the jobs are submitted a scheduler process provides the queue management and scheduling services. The scheduler periodically sends ClassAds of the jobs that are in its queue and have not yet been matched with a resource. The job ClassAds contains information about the requirement of a job such as the amount of physical memory needed by a job, amount of computational time needed by the job, operating system needed to run the job etc. These job ClassAds are also collected by the Condor Collector process at the broker.

The resource brokering service is provided by the broker using the Condor Match Making Service [CONDOR-MMS]. In the Match Making process the attribute specified in the job's ClassAd are parsed and match with corresponding attributes in the ClassAds of resources. A resource which can satisfy all the requirements of the job is then selected as a *match* to run the job. A process called *Condor Negotiator*, also running at the resource broker performs this Match Making. Figure 3.2 shows examples of simple Job and Resource class ads. The job requirements in the job ClassAd says that the job requires a resource with "INTEL" architecture with an operating system of "LINUX" and has disk greater than "6000".



Job ClassAd	Machine ClassAd
MyType = "Job"	**MyType** = "Machine"
TargetType = "Machine"	**TargetType** = "Job"
Requirements = ((other.Arch=="INTEL" && other.OpSys=="LINUX") && other.Disk > my.DiskUsage)	**Machine** = "nostos.cs.wisc.edu"
Rank = (Memory * 10000) + KFlops	**Requirements** = (LoadAvg <= 0.300000) && (KeyboardIdle > (15 * 60))
Cmd = "/home/tannenba/bin/sim-exe"	**Rank** = other.Department==self.Department
Department = "CompSci"	**Arch** = "INTEL"
Owner = "tannenba"	**OpSys** = "LINUX"
DiskUsage = 6000	**Disk** = 3076076
	Department = "CompSci"
	Memory = 512

Figure 3.2: Simple Job and Resource ClassAds

Once matched, the negotiator sends the information about the resource (the execution site) to the scheduler at the submission site where the job was submitted. The scheduler then spawns a process called *grid manager* for the job which remains alive through out the life of the job. The grid manager then submits the job request to the gatekeeper running at the execution site using the GRAM protocol. The gatekeeper in turn launches a Globus job manager at the head node of the execution site. The grid manager keeps track of the job until it has finished, by periodically polling the job manager at the execution site. After the job finishes at the execution site, the standard output and input stream and other log files produced by it are transferred back to the submission site and are stored on a local disk there. These files can be accessed through the Web, if the client does not have access to the machine hosting the submission site. The global data handling services of SAM are not used in the grid level job management but are used in the layer below i.e. the Fabric layer.

3.2 The Fabric Layer and the Grid-to-Fabric interface in SAM-Grid

Fabric is a collection of all the physical computational and storage resources that are a part of the grid infrastructure [FABRIC-1]. The computational resources are mostly clusters of computers that are managed by a local batch system. As described in the last section the grid layer provides a grid level job management, which enables the invocation of a job manager through the gatekeeper on the head node of the execution site. But in order to make actual use of the computational resources the job manager needs to submit *local jobs* to the batch system and perform some other operations at the head node. In SAM-Grid architecture the gap between the grid layer and the Fabric is bridged by a *Grid-to-Fabric Interface* that provides the grid layer with a set of services to enable local job submission and monitoring. The information in the rest of this section has been derived form [FABRIC-1].

3.2.1 The Need for Grid-to-Fabric Interface

A typical Virtual Organization has a lot of computational resources that are managed by different batch systems. There are a plethora of batch systems and batch system configurations in common use that are very different from one another. They differ in the way the jobs are submitted to them, standard output/error streams are specified, jobs are monitored, the input files are staged etc. In order for a grid to incorporate all the resources of a Virtual Organization, the differences between the various batch systems must be abstracted from the grid layer. The way this abstraction is provided in the Condor-G system is by having a separate *job manager* for each batch system and installing the appropriate job manager at each execution site. To incorporate a new batch system a new job manager needs to be written for it. The job managers that come bundled with the Condor-G system are very simplistic in the sense that they only implement interfaces to submit, kill and poll local jobs. These simplistic job managers have been found inadequate for SAM-Grid requirements as discussed below.

The applications being run on the SAM-Grid infrastructure are closely integrated with the data handling system SAM. Thus job submission to the batch system (even an interactive job submission) is done through an interface that allows for pre-submission steps like triggering a SAM project at the head node (gateway node) [FABRIC-2]. There is a need for more sophisticated job managers that can perform some application specific operations at the head node. In addition to this, to facilitate the execution of these applications there is a need to provide an environment at the worker nodes (where the local jobs will eventually run) in which the applications can talk with the Metadata repository, the SAM station at the head node and perform other SAM related operations. In principle this can be done by having the necessary APIs available at the worker nodes using a shared files system. But this imposes the requirement that every execution site needs to have a shared file system which may not always be the case.

A typical grid job results in a large number of jobs being submitted to the local batch system. In other words a typical grid job will map to a number of batch system jobs or local jobs. This number can vary from hundreds at most of the sites to thousands at some sites. The number of local jobs to be submitted is a characteristic of the execution site where the grid job is scheduled. This number depends on the processing capabilities of the cluster which includes the size of the cluster, priority at the cluster etc. This number cannot be determined until the job is scheduled at an execution site and therefore cannot be specified from client side. The job managers should be able to determine the number of local jobs to create for a grid job by either reading the site configuration or by other means which needs knowledge of the applications being run.

The management of job files is another aspect not dealt by the grid technologies in use today. Job files are created at the head node by the execution of a local job. These files include, the standard output/error files of the local job, any log files produced by the local job, any input file that the local job will use etc. 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 shipped back to the client machine to enable the user to determine the outcome and debug problems. So there is a need to track all the job files created by a grid job. Also once a grid job finishes it is necessary to ensure proper clean up to prevent the disk space from filling up.

When batch systems are used in conjunction with a grid, some problems are exposed which may be acceptable in an interactive mode but not in the grid mode. For example batch system commands often suffer from transient failures either due to heavy load on batch system servers or transient network problems. This might be acceptable to an interactive user who can simply reissue the command after a few minutes and continue working, in a grid scenario it will cause the job to fail needlessly. This issue is discussed in detail in the next chapter. The grid middleware should be able to protect itself from such failures as much as possible.

Also the monitoring provided by the standard grid tools is limited to the status of the grid job i.e. if the grid job is active, finished, failed etc. To give the users better indication of the progress of a grid job, we need to provide information about the progress of the local jobs. This includes information about number of local jobs submitted, jobs running, jobs queued, the output files produced by a local job etc.

The grid computing technologies like Condor-G and Globus do not address the local job management requirements described above. These requirements can be met by rewriting the existing job managers that come bundled with Condor-G. However it will make them extremely complex and difficult to deploy. By providing a layer of abstraction above the local batch system, sophisticated job managers with knowledge about the applications being run can be developed easily. This eases the deployment of the grid middleware considerably. In SAM-Grid architecture this is done by providing a set of services at the boundary between the grid layer and the fabric layer. These services together constitute the Grid-to-Fabric interface.

3.2.2 Grid-to-Fabric Interface Design

Figure 3.3 illustrates the design of the Grid-to-Fabric interface in SAM-Grid architecture. The grid layer interfaces with the SAM-Grid job managers. SAM Batch Adapters and Batch system idealizers completely abstract the underlying batch system from the job managers. JIM Sandboxing provides a job file management service to the job managers. The XML based monitoring service lets the job manager collect more information about the progress of a job. In the rest of this section an overview of SAM-Grid job managers and Jim Sandboxing is provided. SAM Batch Adapters and Batch System Idealizers are discussed in detail in the next chapter.

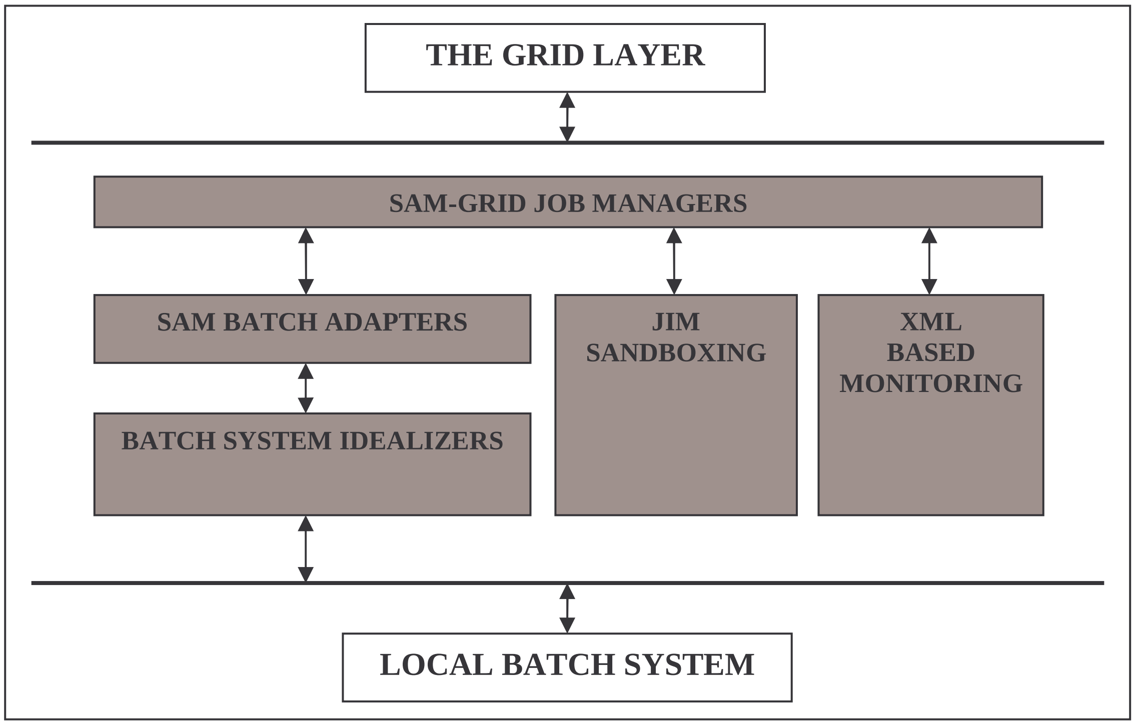


Figure 3.3: Grid-to-Fabric Interface Design

3.2.2.1 JIM Sandboxing

JIM Sandboxing provides a local file management service to the SAM-Grid job managers. 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 as 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 to it. Also if two grid job produce a file with same name it will interfere the execution of the grid jobs 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 their cleanup.

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 at the execution site which is the unique working area for a grid job. All the job files produced by the local jobs belonging to a grid job are deposited in the sandbox area for the grid job. The job managers can instruct the batch system to create the standard output and error files for a local job in its sandbox area.

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. In SAM-grid context the input sandbox is a collection of non-SAM files such as a user specific configuration files. The user supplies the input sandbox at the time of grid job submission in the form of a compressed UNIX tar file. This file is transferred to the head node through Condor-G and is unpacked in the sandbox area of the grid job. At the time of local job submission the job managers add some other files into the sandbox area such as SAM and other product configuration files, the grid credentials (grid proxy) of the user etc.

Once the job manger has initialized the sandbox area for a grid job it then *packages* 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 node and launches user application through a bootstrapping mechanism described in the next chapter. This bootstrapping mechanism also enables the setup of a uniform SAM environment at the worker nodes and to install products or APIs at the worker nodes on the fly rather than requiring them to be present at the worker nodes. Again this is discussed in detail in the next chapter.

JIM Sandbox has a Python API and a command line interface to enable the creation of a sandbox, addition of files to a sandbox, packaging of a sandbox, and to returning the job files in the sandbox in the form of a compressed UNIX tar file.

3.2.2.2 SAM-Grid Job Managers

The SAM-Grid job managers provide the service of grid job instantiation at the execution site by means of mapping a logical grid job definition to a set of local jobs submitted to the batch system [FABRIC-1]. The SAM-Grid job managers conform to the standard Globus GRAM protocol and hence can be invoked through the gatekeeper running at the head node of the execution site. As described earlier the standard grid tools like Condor-G have a different job manager for each batch system. In SAM-Grid architecture there is a single job manager for all the batch system. Instead of the job manager the lower level components (particularly the batch adapters and the batch idealizers) in the Grid-to-Fabric interface provide the logic to abstract the underlying batch system. Figure 3.4 shows the flow of control for job submission in the SAM-Grid job managers.

In SAM-Grid each job is categorized into a *job type* depending on the application being run by the job. Currently there are four job types: *D0 Monte Carlo* for running Monte Carlo simulations for the DZERO experiment, *D0 Merge* to merge the files produced by a job of type D0 Monte Carlo, *D0 Reconstruction* for running the DZERO reconstruction application, and *Mcfarm* for running Monte Carlo simulations using McFarm [MCFARM].

At the grid level, a job is defined logically as a set of parameters; for example a DZERO Monte-Carlo job is defined as a Request Id, the code version to use for production, the input SAM dataset name and the number of events to be simulated. When a job is submitted to the execution site this set of input parameters is transferred to the SAM-Grid job managers through the gatekeeper. Once invoked the job manager triggers local job submission at the execution site. Firstly the job manager processes the input parameters and determines the number of jobs that need to be submitted to the local batch system. This in part depends on the processing capabilities of the site at hand and is read form a local configuration file. Next the job manager initializes a sandbox area for the grid job using the sandboxing interface described earlier. The sandbox area forms the working area for the job manager and all the remaining processing takes place from under this area.

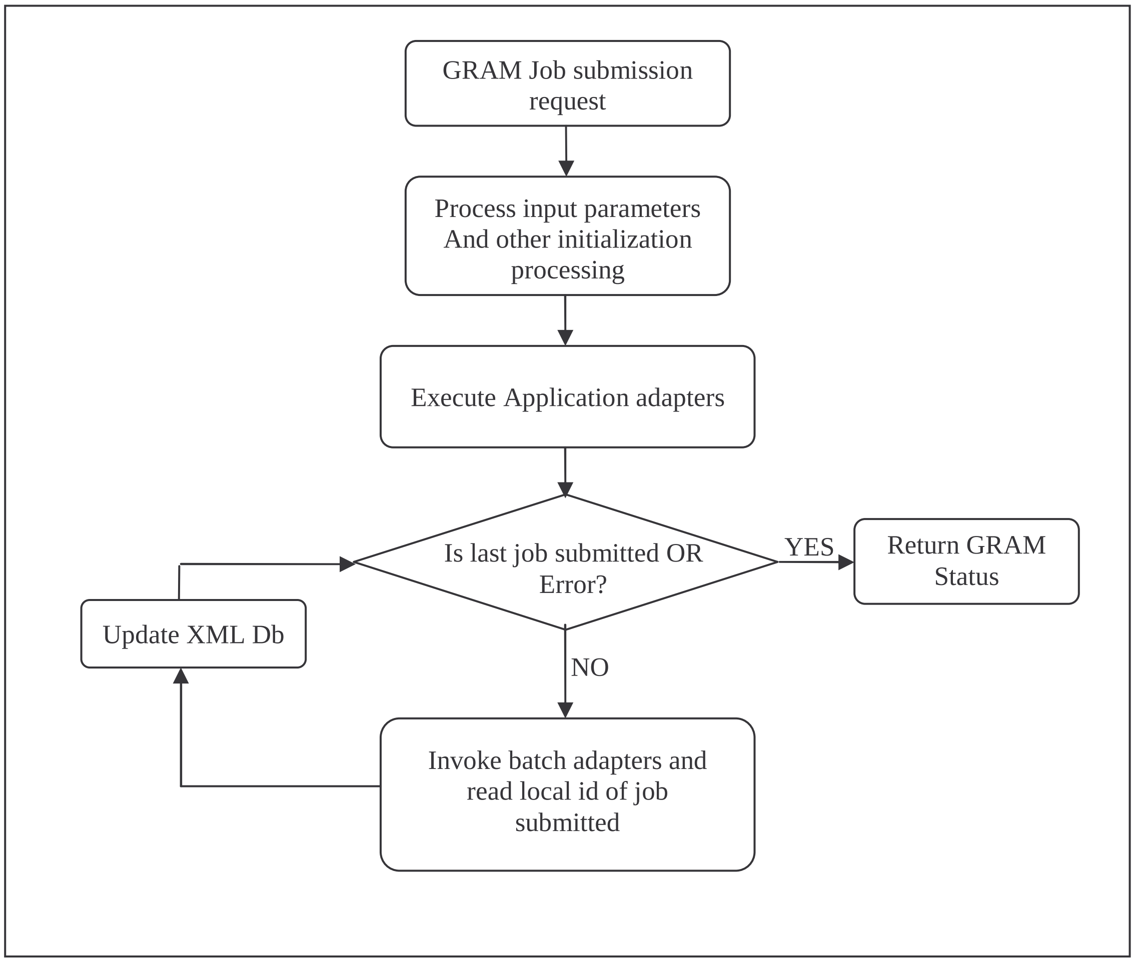

GRAM Job submission request
Process input parameters And other initialization processing
Execute Application adapters
Is last job submitted OR Error?
YES
Return GRAM Status
NO
Update XML Db
Invoke batch adapters and read local id of job submitted

Figure 3.4 SAM-Grid Job Managers flow control

The local job submission results in a set of operations being performed at the head node that depends on the job type of the grid job. In order to support different job types the SAM-Grid job managers support the concept of *application adapters* which are application specific components and perform the pre-submission operations for the application. For example the DZERO Monte-Carlo application adapter downloads the details about the request being processed from the Metadata repository using the SAM data handling system. Some other application such as Reconstruction might require a

SAM project being started at the head node. The application adapters are a very thin or small part of the job managers and hence the application specific aspect is kept to a minimum.

Next the job manager invokes the job submission process by using SAM batch adapters and Batch System Idealizer. It is important to maintain a mapping between a grid job and the local jobs so that the status of a grid job can be tracked. This mapping is created by using the unique grid id assigned to the grid job and the batch idealizers. This is discussed in detail in the next chapter. Essentially during job submission the job managers provide the id of the grid job to the batch idealizers which are responsible for creating the mapping. The batch idealizers in turn return to the job manager the id of the local job submitted. In order to facilitate more precise monitoring, the job manager uses the XML monitoring interface to create an entry in an XML database for each local job submitted with the local id as the key. During this flow if any errors are encountered the job manager returns the appropriate GRAM status back to the grid manager at the submission site.

Once the job submission is complete, the job manager is then responsible for returning the correct status of the grid job to the grid manager running at the submission site. The grid manager periodically sends a GRAM poll request to the execution site to determine the status of the grid job. This request is received by the job managers at the execution site. A grid job is considered to be active as long as there at least one active or queued local job belonging to the grid job in the batch system. In order to determine the status of the grid job the job manager invokes the batch idealizers supplying them the grid id. The batch idealizers return to the job managers the list of corresponding local jobs and their batch system status. The job manager then performs what is termed *status aggregation* to determine the status of the grid job. Status aggregation is the analysis of the batch idealizer output to determine the grid status. The job managers also update the XML database with the current status of each local job. If it is determined that the grid job has finished the job manager collects the output files and the log files from the sandbox area and transfers them to the submission site using GRAM. It then triggers the cleanup of sandbox area for the grid job and finally returns the appropriate GRAM status back to the grid manager. The job manager also provides an interface to terminate a grid job which when invoked (through GRAM) results in the local jobs at the batch system level being terminated or killed.

3.3 SAM-Grid Complete Architecture

Figure 3.5 taken from [CHEP04] shows the complete SAM-Grid architecture with the grid layer, the fabric layer and the SAM data handling system. The SAM data handling system is divided in the grid and fabric layers as well as shown. The global data handling services of SAM are mapped to the grid layer. While the local components such as the SAM Station, the station cache etc. are mapped to the Fabric layer.

The global services of SAM are accessed at various phases during the execution of a job. The job manager uses it to perform some application specific operations on the head node before local job submission is invoked. The global services are also accessed from the worker nodes (where the local jobs run) to perform staging of application executables and the input that it needs.

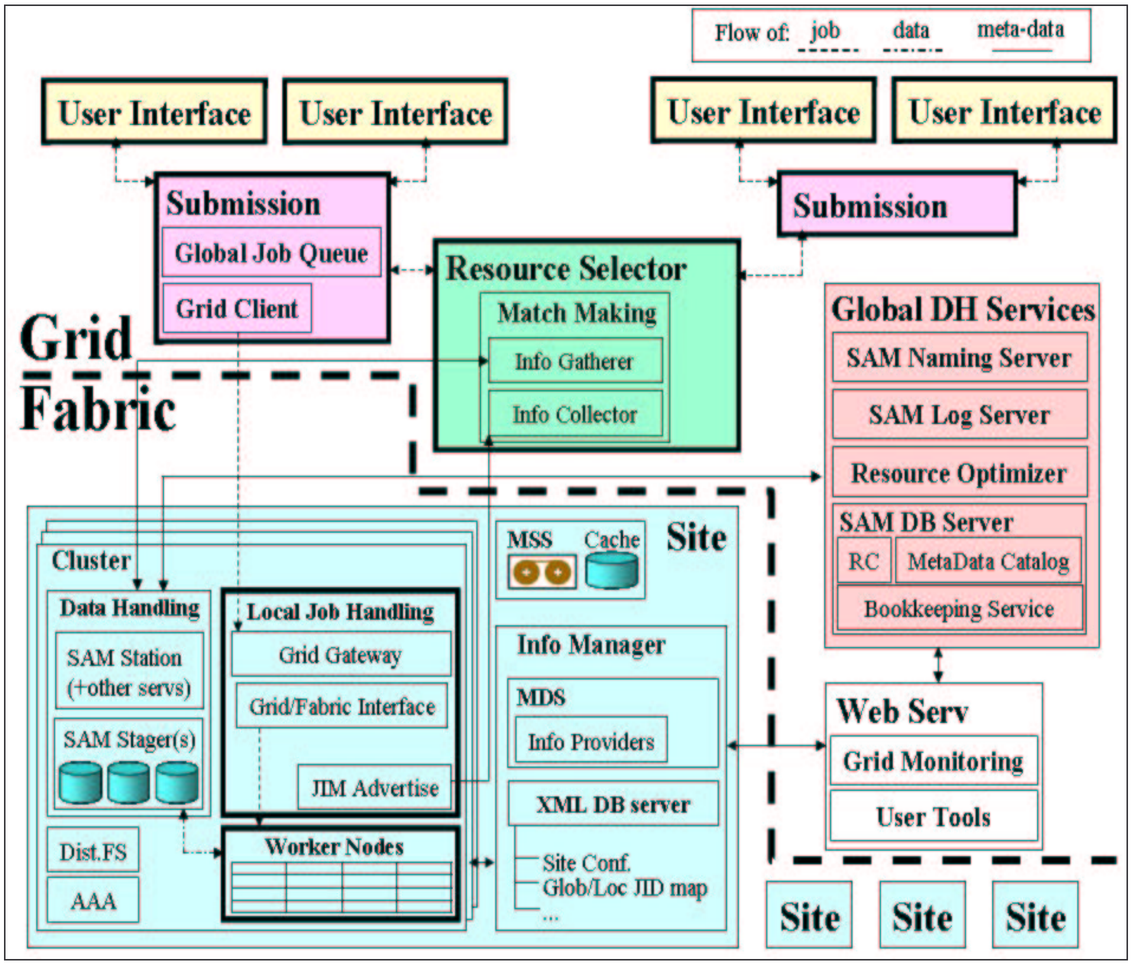


Figure 3.5 SAM-Grid Architecture

The database that is used for storing the monitoring information is Xindice [XINDICE] which is a native XML database. The database can be queried remotely using XML-RPC [XMLRPC] and Tomcat servlet engine [TOMCAT] running at the execution site. The monitoring information stored in the XML database is made available to the grid user using the global Web server. When a user requests the status of a job it results in the Web server querying the XML database at the execution site and returning the status to the user browser. The monitoring database is updated form the worker nodes as well with information about the resource usage and the files produced by a job.

The job manager uses the SAM Logging service to log the details about its operations. These log files are persistently stored on the MSS and can be viewed through the web enabling easy troubleshooting and debugging.

CHAPTER 4

BATCH SYSTEM ABSTRACTION AND IDEALIZATION

As discussed in the previous chapter the SAM-Grid job managers depend on the lower layers in the Grid-to-Fabric interface to provide an abstraction of the underlying batch system and its configuration. The job managers need a uniform interface to -

- Submit jobs to any batch system specifying the executable, the input and the directory path under which the output files produced by the local job should be returned by the batch system
- Determine the current status of a grid job. This means given a grid id the job managers need to know the list of local jobs and their status, that were created as part of the submission of the grid job
- Kill either a single batch job or all the batch jobs belonging to a grid job

This uniform interface is provided to the job managers through the SAM Batch Adapters and the Batch System Idealizers. The implementation of these two components is discussed in detail in this chapter. In addition to this there is a need to create a uniform running environment at the worker nodes where the actual processing takes place, in which the applications can expect a set of services to be available to them irrespective of pre-installed software at the site. This chapter also discusses how

JIM Sandboxing mechanism is used to bootstrap the execution of High Energy Physics applications.

4.1 SAM Batch Adapters

SAM Batch Adapters is a package developed at Fermilab as part of the SAM project. It was originally intended to provide the correct interface to a SAM station for submitting jobs to the underlying batch system [FABRIC-1]. The SAM-Grid architecture has adopted this package as a configuration tool that provides the job manager with interfaces to interact with the batch system. The interfaces themselves are implemented in the Batch System Idealizers which are described in the next section. The SAM Batch Adapter package has a lot of features but here we discuss the aspects of the package that are relevant within the SAM-Grid architecture. For a more detailed reading on the topic the reader can refer [BATCH-ADAPTERS]. The information in the rest of this section has been derived form [BATCH-ADAPTERS].

The SAM Batch adapter package is implemented in Python and has a command line interface available and also provides a Python API. The package is fully configurable and does not make any assumptions about the underlying batch system. Thus it can handle any batch system. The configuration of the package is stored in a local python module which can be updated using an administrative interface the package provides. Figure 4.1 shows part of a typical SAM Batch Adapter configuration.

Each command stored in the configuration has a *command type* associated with it. For example the command shown in figure 4.1 has type “job lookup command”. The

```
batchCommandResult_1 = BatchCommand.BatchCommandResult(0, "", "Success")
batchCommandResult_2 = BatchCommand.BatchCommandResult(0, "%__BATCH_JOB_ID__", "")
batchCommandResult_2 = BatchCommand.BatchCommandResult(1, "", "Failure")
batchCommand_1       = BatchCommand.BatchCommand("job lookup command",
                        "qstat %__BATCH_JOB_ID__”,"[batchCommandResult_1,
                       batchCommandResult_2, batchCommandResult_3,]")
```

Figure 4.1: Part of a typical SAM Batch Adapter Configuration

command types that are used in SAM-Grid are – *job submit command*, *job kill command*, and *job lookup command*. The function of these command types can be derived from their name. 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 can execute to get the desired results. For example in figure 4.1 the command string for the job lookup command is “qstat %__BATCH_JOB_ID__”. The user or the client can read the command string giving its command type and then replace the template string which in this case is “%__BATCH_JOB_ID__” with the id of a local job and then execute the resulting command to perform lookup on a single batch job. The use of string templates in SAM-Grid is discussed later in this section.

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 4.1 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 the batch job id. The third result states that an exit status of 1 means that the command has failed.

It is important 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 user to execute commands and interpret their results. The commands shown in the examples in this section have embedded batch system commands in them. However in SAM-Grid the batch system functionalities are provided within Batch System Idealizers for reasons explained in the next section. Thus in SAM-Grid the command strings in SAM Batch Adapter configuration contain the idealizer scripts rather than batch system commands. For example the command string for a look up command looks something like "sam_pbs_handler.sh job_lookup --project=%__USER_PROJECT__ --local-job-id=%__BATCH_JOB_ID__". The idealizer script in this case is sam_pbs_handler.sh which provides the interface to PBS batch system.

As mentioned earlier there are a lot of string templates defined in SAM Batch Adapter, but only a few are used in SAM-Grid. The rest of this section lists the template strings used in SAM-Grid and their meaning in SAM-Grid context.

- %__USER_PROJECT__: It was originally intended to provide a name to a SAM Project, but in SAM-Grid it is used to specify the grid id of a job to the idealizer scripts.
- %__USER_SCRIPT__: It is used to specify the name of the executable to be submitted to the local batch system.
- %__USER_SCRIPT_ARGS__: It is used to specify the arguments if any, to the executable submitted to the local batch system
- %__USER_JOB_OUTPUT__: The path where the standard output file of the batch job should be deposited.
- %__USER_JOB_ERROR__: The path where the standard error file of the batch job should be deposited.
- %__BATCH_JOB_ID__: To specify the id of a single batch job.
- %__BATCH_JOB_STATUS__: The current status of a batch job.

4.2 Batch System Idealizers

Batch System Idealizers together with SAM Batch Adapters provide a complete abstraction of the underlying batch system to the job managers. Batch System Idealizers implement the interfaces required to perform batch system operations such as submitting jobs, checking the status of a job etc. SAM Batch Adapter is just an interface to invoke the idealizer scripts. Batch System Idealizers as the name implies, idealize the batch systems to make their interactions with the grid machinery easier by mitigating any imperfections and adding any missing features [FABRIC-1]. The idealizer scripts are totally batch system specific and a new batch system can be added to the grid infrastructure by simply writing an idealizer script for the batch system.

4.2.1 Need for Batch System Idealizers

This section discusses the rationale behind the Batch System Idealizers. We can use the interfaces provided by the batch system itself along with SAM Batch Adapters to provide an abstraction of the batch system. However there are a lot of problems that are exposed when the batch system interfaces are used in a grid scenario.

One problem that almost all the batch systems suffer with is the transient failures in executing some commands. While these transient failures may be acceptable to an interactive user, in a grid scenario they will cause the grid job to fail. For example if the execution of a batch system polling command fails due to a network glitch or the command times out because of heavy load on the server, the job managers will fail to report the correct status back to the grid job manager at the submission site, which will falsely interpret the job as a failure and proceed with clean up operations. 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 (in the order of thousands) of jobs running in the batch system which is a common occurrence in a grid scenario. Such transient failures can be avoided by simply retrying over a period of few minutes.

The output produced by a batch system command needs to be parsed by the job managers so they can interpret the results. However the output produced by commands in different batch systems differs from each other. In some cases the output maybe too verbose, making it really complex to parse it. Also the status of a batch job is represented in different ways in different batch systems. For example some batch system represent the status of a running job simply as *running* while some batch systems may call it *active*. Thus there is a need to map the batch system specific status of a job to a set of standard status that the job managers understand.

There is a need to create a mapping between a grid job and the local jobs that were submitted as part of the grid submission. This mapping is created in a totally batch system specific way. For example in some batch systems this can be done by setting the name of the local job to the id of the grid job. In general the way this mapping is created varies from one batch system to another. There is a need to abstract the creation of this mapping from the job managers.

In a cluster every worker node has a certain amount of scratch space reserved for a local job that serves as its working area. 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. Not all the 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 a full fledged scratch management support while some other batch systems like PBS do not have any support for scratch management. There is a need to transparently provide scratch management support for batch systems that do not provide this service.

Another problem that is prevalent in cluster computing is what we term as the *Black Hole Effect*. 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. This results in the faulty node acting like a *black hole* and 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 fails the job in less than a minute and becomes free again. 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 is repeated. If the jobs are continuously dispatched to the same node it will result in only 9 out of the 100 jobs finishing successfully. A success rate of just 9% is unacceptable. Common examples of faults that result in the Black Hole Effect are – a faulty network interface at the node resulting in files getting corrupted, DNS miss-configurations etc. An interactive user can 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.

The problems that are identified above are solved by implementing the batch system interfaces in the batch idealizers as discussed in the next section. The batch idealizers abstract the client (job manager) from these problems giving them a picture of a *grid friendly* batch system.

4.2.2 Implementation of BS Idealizers

Most of the batch adapters in SAM-Grid are written in UNIX shell scripts and a few are written in Python. The batch systems that are currently supported by SAM-Grid i.e. the batch systems for which an idealizer exists are – The batch at CC-IN2P3 (BQS) [BQS], the Portable Batch System [PBS], the Condor Batch System [CONDOR-URL] and Farms Batch System Next Generation [FBSNG].

In order to overcome the problems with transient failure in batch system commands retrials are incorporated with every batch system command in the batch idealizers. The time interval for these retrials is configurable, but for them to be effective must span be in the order of minutes. This is because the mild failures these retrials are supposed to mitigate should disappear in a few minutes. If the problem is severe and lasts more than the retrial interval then its 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 *id* associated with the grid job and associate it with the local job in a batch system specific way. The way it is done for different batch system is discussed later.

To provide a uniform interface of the batch system to the job managers 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. They also perform a mapping of the batch system status to a set of status that the job managers understand. The statuses that are currently supported are: *active*, *failed*, *suspended*, *pending*, and *submitted*. As an example if a batch system reports the status of a job as *queued* the batch idealizers will report its status as *pending*.

The batch idealizers also provide scratch management support for batch systems that do not already do so. This feature is provided by writing special scratch management scripts that only the batch idealizers know about thus abstracting this limitation from the job managers. The scratch management scripts are staged to the worker nodes using the batch system and form the 1st stage of execution. The path of the scratch area at worker nodes is read from configuration at the head node. The scratch management scripts create a separate work area or directory for the local job under this path. They then launch the user executable from this unique scratch area. After the job finishes they clean up the directory associated with job and return the appropriate exit status to the batch system.

The batch system idealizers provide a partial solution to the Black Hole Problem. While solving this problem in an automated way is non-trivial, the batch idealizers soften its blow by maintaining a *neglect list*, which contain the names of the nodes known to have such problems. While job submission they explicitly ask the batch system not to schedule jobs to nodes in the neglect list. At this point this list is being maintained manually and whenever a problem like above is identified the site administrator will need to update this list. This solution is incomplete because it still requires some human intervention. 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.

4.2.2.1 Batch System Idealizer Interface

Figure 4.2 shows the interface of the batch idealizers in SAM-Grid. The batch idealizers need not have the same interface because they are invoked through SAM Batch Adapters. However a uniform interface makes configuration of SAM Batch Adapter lot easier and hence it is desired.

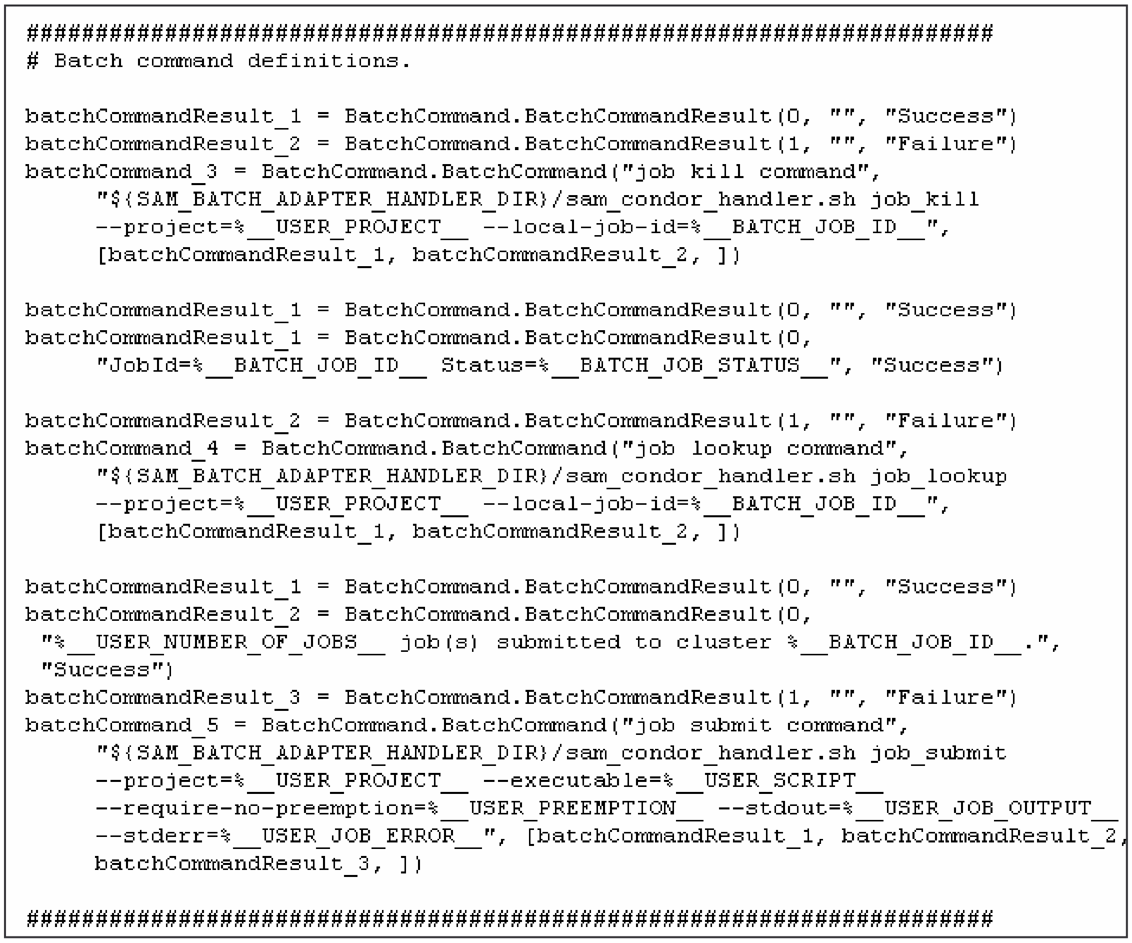
```
           ACTIONS SUPPORTED BY ALL IDEALIZERS

For submitting a job :  job_submit
For polling a job    :  job_lookup
For killing a job    :  job_kill

           ARGUMENTS ACCEPTED BY ALL IDEALIZERS

--executable=%__USER_SCRIPT__
--project=%__USER_PROJECT__
--arguments=%__USER_SCRIPT_ARGS__
--stdout=%__USER_JOB_OUTPUT__
--stderr=%__USER_SCRIPT_ERROR__
--local-job-id=%__BATCH_JOB_ID__


                    COMMAND FORMATION
idealizer_script <action> <arguments>
```

Figure 4.2: Batch Idealizer Interface

As shown in figure 4.2, all the batch idealizers accept three actions.

1. *job_submit*: To submit a job to the batch system.
2. *job_lookup*: To check the status of job(s).
3. *job_kill*: To kill batch job(s).

The arguments are supplied to the idealizers using the concept of template substitution described in section 4.1. The meaning of each argument can be derived from the template string description given in section 4.1.

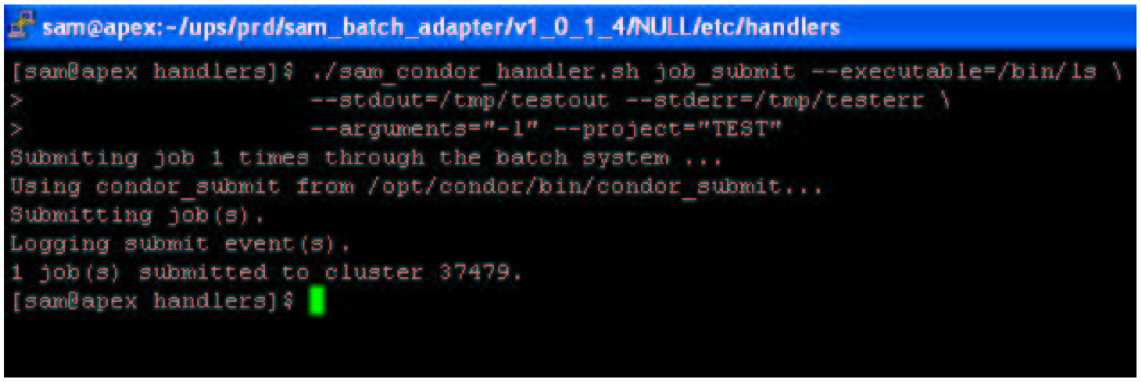


```
#########################################################################
# Batch command definitions.

batchCommandResult_1 = BatchCommand.BatchCommandResult(0, "", "Success")
batchCommandResult_2 = BatchCommand.BatchCommandResult(1, "", "Failure")
batchCommand_3 = BatchCommand.BatchCommand("job kill command",
     "${SAM_BATCH_ADAPTER_HANDLER_DIR}/sam_condor_handler.sh job_kill
     --project=%__USER_PROJECT__ --local-job-id=%__BATCH_JOB_ID__",
     [batchCommandResult_1, batchCommandResult_2, ])

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, ])

batchCommandResult_1 = BatchCommand.BatchCommandResult(0, "", "Success")
batchCommandResult_2 = BatchCommand.BatchCommandResult(0,
 "%__USER_NUMBER_OF_JOBS__ job(s) submitted to cluster %__BATCH_JOB_ID__.",
 "Success")
batchCommandResult_3 = BatchCommand.BatchCommandResult(1, "", "Failure")
batchCommand_5 = BatchCommand.BatchCommand("job submit command",
     "${SAM_BATCH_ADAPTER_HANDLER_DIR}/sam_condor_handler.sh job_submit
     --project=%__USER_PROJECT__ --executable=%__USER_SCRIPT__
     --require-no-preemption=%__USER_PREEMPTION__ --stdout=%__USER_JOB_OUTPUT__
     --stderr=%__USER_JOB_ERROR__", [batchCommandResult_1, batchCommandResult_2,
     batchCommandResult_3, ])

#########################################################################
```

Figure 4.3: SAM Batch Adapter Configuration

Figure 4.3 shows how batch idealizers are used in conjunction with the SAM batch adapters. When the SAM batch adapters are configured at a particular site the command strings include the path to the batch idealizer script for the batch system at the site. Figure 4.3 shows such a configuration for the Condor batch system. The job managers need to read the command using SAM batch adapters and the substitute the arguments in place of template strings and execute the batch idealizer scripts.

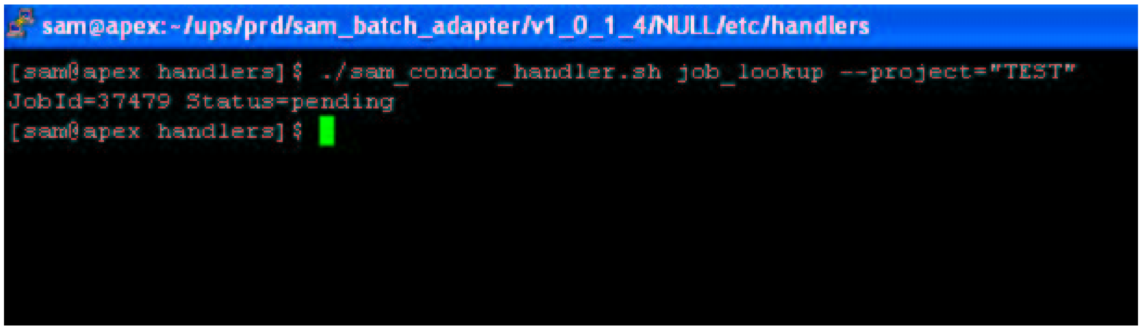
Figure 4.4 shows the job submission to a Condor batch system. The job managers will search for the expected output string (see figure 4.3) in the output of the batch idealizer and determine the local job id of the batch job.



```
sam@apex:~/ups/prd/sam_batch_adapter/v1_0_1_4/NULL/etc/handlers
[sam@apex handlers]$ ./sam_condor_handler.sh job_submit --executable=/bin/ls \
>                    --stdout=/tmp/testout --stderr=/tmp/testerr \
>                    --arguments="-l" --project="TEST"
Submiting job 1 times through the batch system ...
Using condor_submit from /opt/condor/bin/condor_submit...
Submitting job(s).
Logging submit event(s).
1 job(s) submitted to cluster 37479.
[sam@apex handlers]$
```

Figure 4.4: Job submission through batch idealizer

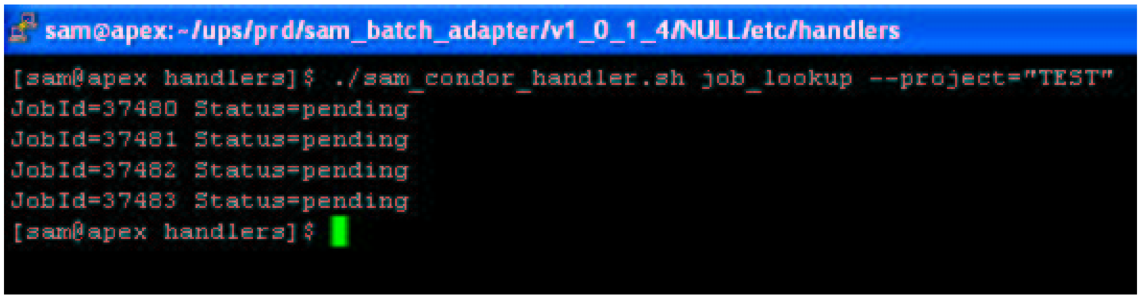
Figure 4.5 shows the lookup of a job through the batch idealizers. In this case there is just a single batch job belonging to a grid job with grid-id “TEST”. The output produced contains the mapped status of the job. Again the job managers can interpret this output using the SAM Batch adapter configuration.



```
sam@apex:~/ups/prd/sam_batch_adapter/v1_0_1_4/NULL/etc/handlers
[sam@apex handlers]$ ./sam_condor_handler.sh job_lookup --project="TEST"
JobId=37479 Status=pending
[sam@apex handlers]$
```

Figure 4.5: Job lookup through batch idealizer

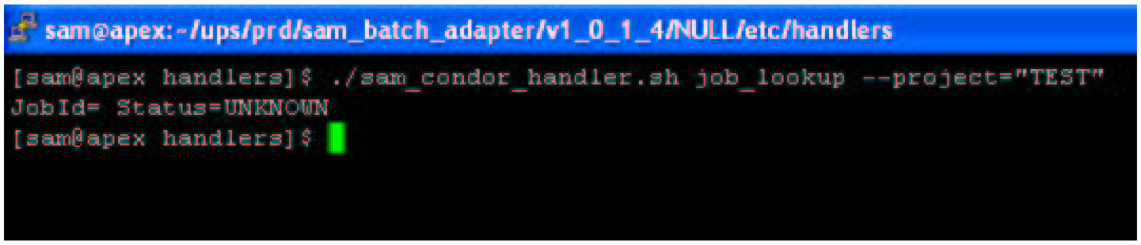
Figure 4.6 shows the job lookup when a grid job spawns multiple local jobs. When updating the status of individual batch jobs, the job manager look for its id in the output produced by the idealizers. If the id is missing from the list, the job has finished and the status is updated.



```
sam@apex:~/ups/prd/sam_batch_adapter/v1_0_1_4/NULL/etc/handlers
[sam@apex handlers]$ ./sam_condor_handler.sh job_lookup --project="TEST"
JobId=37480 Status=pending
JobId=37481 Status=pending
JobId=37482 Status=pending
JobId=37483 Status=pending
[sam@apex handlers]$
```

Figure 4.6: Job lookup for multiple local jobs belonging to a single grid job

Figure 4.7 shows the output of lookup when all the batch jobs belonging to a grid job have finished. At this point the job managers can aggregate the grid job status as *done*. If during lookup job managers find even a single batch job belonging to a grid job the aggregated status of that grid job will either be *active* or *failed* depending on the status of the batch job.


sam@apex:~/ups/prd/sam_batch_adapter/v1_0_1_4/NULL/etc/handlers

```
[sam@apex handlers]$ ./sam_condor_handler.sh job_lookup --project="TEST"
JobId= Status=UNKNOWN
[sam@apex handlers]$
```

Figure 4.7: Output of lookup when the job has finished

The rest of this section discusses the batch system specific aspects in the implementation of different batch system idealizers.

4.2.2.2 Batch System Idealizer for Condor

The batch system idealizer for Condor is implemented as a UNIX shell script called sam_condor_handler.sh. Job submission to Condor requires the creation of a job description file (JDF). sam_condor_handler.sh script creates this JDF at the time of job submission before invoking the batch system. The mapping of the grid-id and local jobs is created by setting a variable called *sam_project* in the JDF to the grid id. When performing lookup, the batch system is asked for only those jobs that were submitted with sam_project variable set to the id of the grid job.

A feature that the condor batch idealizer supports and is not supported by any other batch idealizer is the pre-emption requirement of the job. Since Condor supports process migration, a job can be evicted from a node it occupies if a higher priority job is submitted. This pre-emption is not acceptable for some type of jobs like the SAM-Grid merging jobs. Such jobs can be submitted to the batch system requiring that once started they should not be pre-empted.

4.2.2.3 Batch System Idealizer for PBS

The batch idealizer for PBS is implemented as a UNIX shell script called sam_pbs_handler.sh. Here the mapping between grid and local jobs is created by submitting the local jobs with their *name* attribute set to the id of the grid job. While performing lookup on a grid id, only jobs having name as the grid id are returned. The value assigned to the name attribute is not exactly the grid id but a unique string extracted from it. This is done because the grid id may violate the length restriction imposed on attribute name by the batch system.

One fallacy of the PBS batch system is the absence of any scratch management. This is overcome by the pbs_scratch_setup.sh script. This scripts and the user executable are transported to the worker through the batch system. Upon its execution the scratch management script creates a unique directory (based on local job id) for the job and then launches the user executable. 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. This problem is eliminated by having the scratch management script examine the scratch area and cleaning up any directories belonging to jobs killed. Thus if the scratch directory for a job is left dangling it will be cleaned when the next job is scheduled at that node.

4.2.2.4 Batch System Idealizer for BQS

BQS is a batch system developed at the IN2P3 computing center in Lyon, France [IN2P3]. It is very similar to the PBS batch system. The batch idealizer for BQS performs mapping between the grid id and local jobs in a manner similar to that in PBS idealizer described in the previous sub-section. BQS has the property that when a number of jobs are submitted they are buffered locally and then jobs are dispatched to the batch system server in bulk. This poses a problem in the grid environment. If the grid job submission finishes and the local jobs (submitted as part of grid submission) are still buffered locally by the time first polling request for the grid job arrives, then the job managers will assume the grid job to be done because the batch jobs have not yet been actually submitted to the batch system server. This problem is overcome in the BQS idealizer. When a job is submitted the idealizer determines if it is the first local job belonging to the grid job. If it is, the idealizer blocks and waits for this job to shows up on the batch system polling command. Once this happens the idealizer then returns the id of the local job and further job submissions proceed normally. If the first batch job does not show up in the batch system queue for a certain amount of time (this time is configurable), then the idealizer returns with an error.

4.2.2.5 Batch System Idealizer for FBSNG

FBSNG is a batch system developed by the Integrated Systems Development Department at Fermilab [ISD]. It is based on Farm Batch System. The idealizer for FBSNG has been implemented as a python script called sam_fbsng_handler.py. It makes use of FBSNG python API for interacting with the batch system. The mapping for grid id to local jobs is provided by setting the section name of the local jobs to the grid id. For more details on FBSNG the reader can refer [FBSNG]. There is a FBSNG scratch management script called fbs_scratch_setup.sh and is similar to one for PBS. The only difference in scratch management here is that the batch system takes care of the cleanup of scratch areas and hence the problem of dangling scratch area is not encountered in case of FBSNG.

4.3 Bootstrapping the Execution of User applications

The SAM-Grid system depends on SAM for its data handling needs. As described earlier the application executables and its input data are staged at the worker nodes via SAM. In order to perform this stage-in at the worker nodes the flow of control passes through multiple stages with each stage performing a set of operations that enable the stage-in in the last bootstrapping stage after which the user application is launched.

Figure 4.8 shows the bootstrapping process at a worker node. The execution at the worker node begins with a self extracting compressed file which is created as part of packaging a sandbox at the head node. It contains within it a GridFTP client, the user's grid credentials i.e. the X509 proxy of the grid user and a control script called sandbox manager. When executed the self extractor untars its contents and passes the control to sandbox manager. The sandbox manager sets up the grid client and then uses GridFTP to copy a set of files from the job's sandbox area at the head node. These files include a compressed tar file that contains SAM APIs & their dependencies, the XML database client API, and configuration files. They also include scripts that form the next stages of bootstrapping. The sandbox manager then sets up these products. The configuration files for different products are created in the sandbox area at the head node at the time of job submission and are derived from the configuration at the head node itself. Hence to enable the bootstrapping of dependencies, the administrator just needs to configure the head node properly. The file transfer mechanism used to transfer these files between the head and the worker nodes is GridFTP. It is very robust and scales well even for hundreds of parallel transfers.

After sandbox manager finishes execution, an environment has been setup in which the lower layers of bootstrapping and the user applications can talk to SAM or use the monitoring database. Thus there is no need of any software to be pre-installed at the worker nodes. Instead all the dependencies are setup dynamically using this bootstrapping procedure. This makes the deployment and administration of the infrastructure very easy.

Next the sandbox manager invokes the next bootstrapping stage which updates the monitoring information for each local job. This stage is implemented as a UNIX shell script called *jim_rapper.sh*. It updates the XML monitoring database with the hostname where the execution is taking place. When the execution of user application

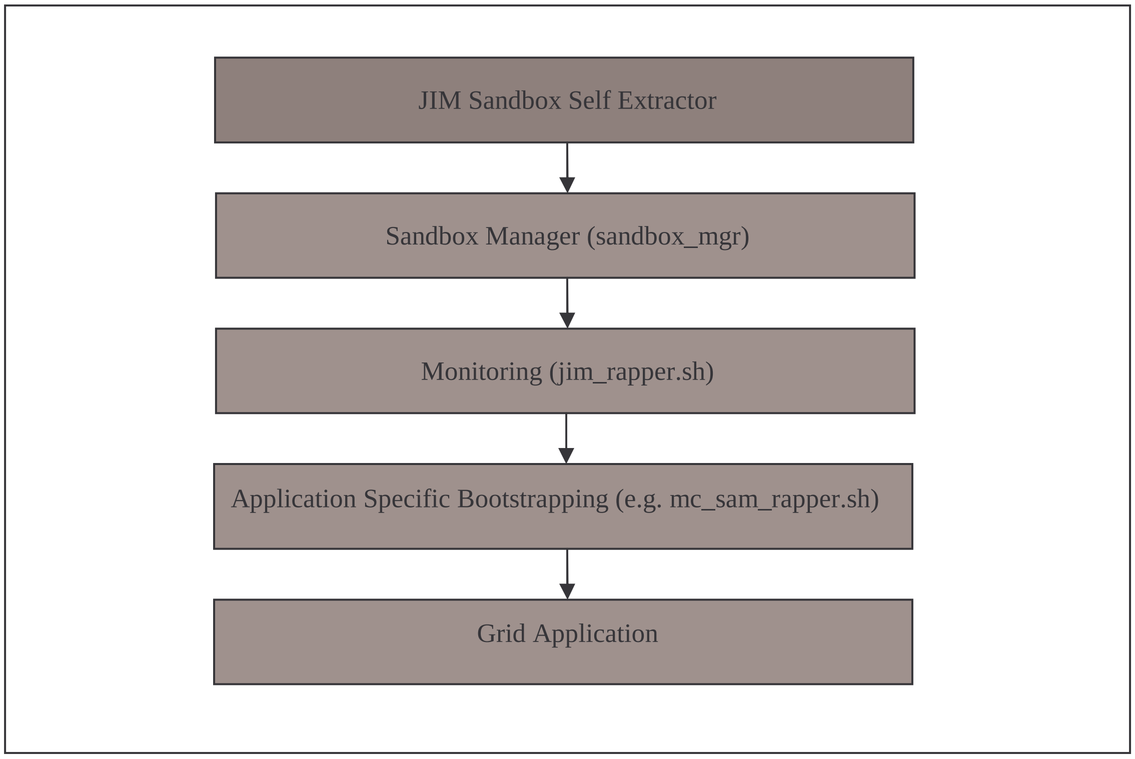
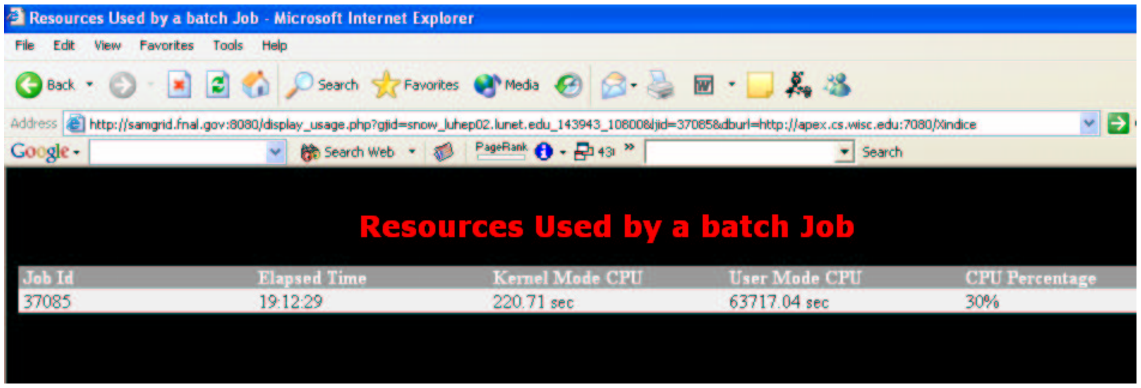

JIM Sandbox Self Extractor
Sandbox Manager (sandbox_mgr)
Monitoring (jim_rapper.sh)
Application Specific Bootstrapping (e.g. mc_sam_rapper.sh)
Grid Application

Figure 4.8: Bootstrapping at a worker node

finishes it updates the database with the usage information (see Figure 4.9). This information is vital to keep track of resource consumption by various Virtual Organizations. Also it stores the information about the job outcome i.e. the exit code of user application.


Resources Used by a batch Job - Microsoft Internet Explorer
File Edit View Favorites Tools Help
Back Search Favorites Media
Address
Google Search Web PageRank 431 Search

http://samgrid.fnal.gov:8080/display_usage.php?gjid=snow_luhep02.lunet.edu_143943_10800&jid=37085&dburl=http://apex.cs.wisc.edu:7080/Xindice

Resources Used by a batch Job

Job Id	Elapsed Time	Kernel Mode CPU	User Mode CPU	CPU Percentage
37085	19:12:29	220.71 sec	63717.04 sec	30%

Figure 4.9: Usage information in XML database

Another feature that is provided by the monitoring stage is protection against a run-away job. A run-away job is a failed job that is not producing any output and is stuck in some sort of endless loop. This feature is implemented in the script called *accelerator.sh*. This is done by keeping track of the output directory of the job and periodically checking if the files under it are getting updated or not. If the files have not been updated for a long time then it means that the job is stuck and is not doing any useful processing. In this case the user application is terminated thus freeing up the node for further processing.

The final stage in the bootstrapping process is application specific. An example of this stage is *mc_sam_rapper.sh* script which is responsible for bootstrapping the DZERO Monte-Carlo application. Here the application input and the application itself are staged from SAM by making use of SAM APIs available at this stage. It also updates the XML database with information about the time taken to perform various staging operations.

4.4 Intra-Cluster transfers in SAM

SAM is a data handling system that provides the connectivity between various SAM stations and MSS. When a user requests a file using SAM, the station gets the file from either another station's cache or the MSS to its own cache. It is then the responsibility of the user to copy the file in its working area. The most common way that SAM has been used so far has been by having the station's cache visible across the entire cluster through a shared file system. The user's scripts/programs then access these files through the shared file system. However in a Grid environment this mode of operation is not acceptable because not all sites will have a cluster-wide shared file system and also copying files over a shared file system doesn't scales well especially when there are hundreds of transfers in parallel.

In order to allow the files to be copied from the SAM station's cache to the worker nodes, SAM-Grid makes use of the *sam_cp* package developed at Fermilab [SAMCP]. The sam_cp product is a wrapper around various file transfer protocols and has traditionally been used by the SAM stations to transfer files between them. The details about of design of sam_cp can be found in [SAMCP-DESIGN]. sam_cp allows the transfer of files by a command like *sam_cp srcFile destFile*. The actual transfer protocol that will be used for the file transfer is configurable. In SAM-Grid the file transfer protocol used for this is GridFTP.

If a site has the capacity to process hundreds of jobs in parallel then there is a risk of the intra cluster file transfers choking the network interface at the machine hosting SAM station's cache. This problem is exacerbated when the size of file being transferred is huge (hundreds of megabytes) which is often the case in case with SAM files. In case the SAM station and the head node are being hosted on the same machine this problem can result in wide spread failures. This is because if the network-interface at the head node is choked then there is a risk of the submission site not being able to communicate with the job managers and thus the grid job can fail. In addition to this the communication with the batch system server might fail as well.

Thus there is a need to limit the number of concurrent file transfers from the SAM cache. In SAM-Grid this is done using Farm Remote Copy utility (FCP) [FCP]. FCP is a product of the ISD group at Fermilab. It restricts the number of parallel file transfer operations from the SAM cache. The maximum number of transfers allowed is configurable. FCP has client-server architecture. Before performing the copy operation the client contacts the FCP server and requests the file transfer. If this request will result in the number of file transfers exceeding the set limit, the server makes the block until one of the transfer finishes. At that point the client is given the permission to proceed with the file transfer. If there is more than one client waiting for permission then the server signals them on a First come First Serve basis. For more details about the design and functioning of FCP the reader can refer [FCP].

In SAM-Grid the FCP server runs on the machine hosting the SAM cache. The FCP and sam_cp clients are setup dynamically at the worker nodes using the bootstrapping mechanism described in section 4.3. In order to make use of FCP in SAM-Grid, we have integrated it with sam_cp and GridFTP. When sam_cp is invoked at the worker nodes it results in FCP client contacting the server. Once the client gets the permission to copy file, FCP invokes GridFTP to perform the actual file transfer.

The maximum number of concurrent transfers that are allowed depends on the site at hand. In particular it depends on maximum number of jobs that can run in parallel at the site and the capacity of the network at the site.

CHAPTER 5

CONCLUSIONS

The SAM data handling system has been in production use since 1999. The SAM-Grid system has augmented the data handling capability of SAM by adding job and information management. This chapter lists the important conclusions that can be drawn from this work. It also provides some statistics about the current usage of this system and then discusses scope for future enhancements to the current infrastructure.

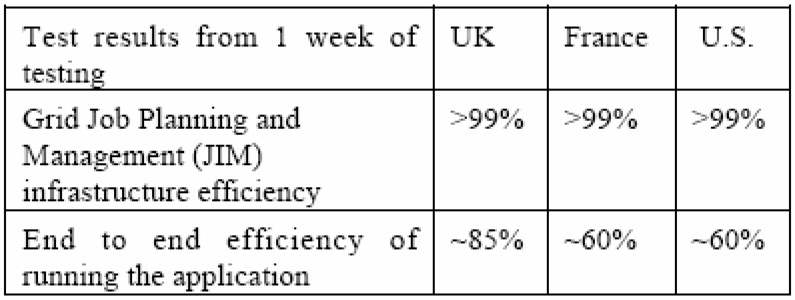
5.1 Conclusions

The work in this thesis identified some key problems that make the integration of batch systems with a grid difficult. Current research in grid community is not addressing the issues put forward in this thesis. By providing a layer of abstraction above the batch systems we have developed a framework in which batch systems can be easily integrated with a grid middleware such as Condor-G.

The SAM-Grid system has enabled the High Energy Physics applications such as Monte Carlo simulations to be run over a grid. The SAM-Grid system provides a distributed resource management for physics collaborations such as DZERO which has resulted in more efficient resource utilization. In addition to this development of sophisticated job managers has resulted in an infrastructure in which new applications can be easily integrated into the grid. An example of this is the integration of McFarm [MCFARM] with SAM-Grid.

5.2 Current utilization of SAM-Grid system

The SAM-Grid system has been deployed for production starting from January 2004. It has since been used to do Monte Carlo event simulation for the DZERO experiment. Through September 2004 the system had produced over 2 million events which is equivalent to about 17 years of computation on a typical GHz CPU [CHEP04]. By adding various idealization features and other enhancements the inefficiency in event production due to the grid infrastructure has been reduced from 40% during early phases in deployment to about 1% [CHEP04]. Figure 5.1 from [PPDG-DOC] lists the efficiency of the infrastructure as observed during a one week stress test period during which the system was tested for robustness by submitting jobs to three test sites. The results were then presented in [PPDG-DOC].



Test results from 1 week of testing	UK	France	U.S.
Grid Job Planning and Management (JIM) infrastructure efficiency	>99%	>99%	>99%
End to end efficiency of running the application	~85%	~60%	~60%

Figure 5.1: Infrastructure efficiency during 1 week test period

The end-to-end efficiency of running the application is less than JIM efficiency because they include the failures of the applications themselves. As can be seen failures due to the grid infrastructure are less than 1 % making this a very reliable and robust system. Figure 5.2 lists some additional numbers collected by the SAM-Grid team to study the end-to-end efficiency of running DZERO montecarlo at three sites over a 6 week period starting from February 02, 2004.

Week	Average Efficiency	Lyon, France	Manchester, U.K.	Wisconsin, U.S.
1	67	56.4	86	58.4
2	65	55.9	77	62.1
3	90	95.3	84.5	91.2
4	87	93.9	85	81.4
5	94	98.05	93.2	91.36
6	95	98.5	94.6	92

Figure 5.2: End-to-end efficiency at 3 sites over a 6 week period

The SAM-Grid system is currently being tested for running Monte Carlo application for the CDF experiment and for running Reconstruction application for both the DZERO and CDF experiment.

5.3 Future Work

In the current infrastructure the administrator needs to manually maintain the list of bad nodes where the application consistently crashes. There is a need to automate the computation of this list in order to avoid the Black Hole effect. Once this is done there will be no need of human intervention for normal operations.

Also the layering in the bootstrapping process can be further enhanced by having a separate layer to perform SAM stage-in operations. Currently this is done as part of the application specific layer.

In the grid layer full-fledged brokering which makes use of information about site usage and capabilities needs to be enabled. Currently the brokering is mainly manual with the user explicitly specifying the destination of a job.

REFERENCES

[FOSTERPHYSICSTODAY] Foster, I., *The Grid: A new infrastructure for 21st Century Science*. Physics Today, Feb 2004

[ANATOMY] Foster, I., C. Kesselman, and S.Tuecke, *The Anatomy of the Grid*. International J. Supercomputer Applications, 15(3), 2001.

[FNAL] Fermi National Accelerator Laboratory, http://www.fnal.gov

[CDF-1] The Collider Detector at Fermilab, http://www-cdf.fnal.gov

[D0-1] DZERO, http://www-d0.fnal.gov/

[ENSTORE] Fermilab Mass Storage System, http://www-isd.fnal.gov/enstore

[SAM-1] 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

[PPDG] Particle Physics Data Grid Home page, http://www.ppdg.net/

[GLOBUS-PAGE] The Globus Alliance home page, http://www.globus.org

[GLOBUS-TOOLKIT] Globus Toolkit, http://www-unix.globus.org/toolkit

[GSI-1] Welch, V., et al. *X.509 Proxy Certificates for Dynamic Delegation* in 3rd annual PKI R&D Workshop, 2004

[GRAM-URL] Globus Resource Allocation Manager Overview, http://www-unix.globus.org/developer/resource-management.html

[GASS] Global Access to Secondary Storage, http://www.globus.org/gass/

[GRIDFTP-URL] Globus GridFTP, http://www.globus.org/datagrid/gridftp.html

[GRIDFTP-DOC] GridFTP Update January 2002, http://www.globus.org/datagrid/deliverables/GridFTP-Overview-200201.pdf

[CONDORG-URL] Condor-G, http://www.cs.wisc.edu/condor/condorg/

[CLASSADS] Condor ClassAd, http://www.cs.wisc.edu/condor/classad

[CONDORG] 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

[SAM-2] Carpenter, L., et al., *SAM and the Particle Physics Data Grid* in the proceedings of Computing in High Energy and Nuclear Physics, Beijing, China, September 2001

[CHEP04] 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

[CONDOR-MMS] Condor Match Making Service, http://www.cs.wisc.edu/condor /manual/v6.0/2_3Condor_Matchmaking.html

[FABRIC-1] Grid to Fabric job submission interface in SAM-Grid, http://www.d0.fnal. gov/computing/grid/SAMGridFabricJobSubmission.html

[FABRIC-2] 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

[XINDICE] Xindice, http://xml.apache.org/xindice/

[XMLRPC] XML-RPC, http://www.xmlrpc.com/

[BATCH-ADAPTERS] SAM Batch Adapters, http://d0db.fnal.gov/sam_batch_adapter/ sam_batch_adapter.html

[BQS] The Batch at IN2P3, http://webcc.in2p3.fr/man/bqs/intro

[PBS] Portable Batch System, http://www.openpbs.org/

[CONDOR-URL] Condor Overview, http://www.cs.wisc.edu/condor/description.html

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

[IN2P3] IN2P3 Computing Center, http://institut.in2p3.fr/

[ISD] Integrated Systems development Department at Fermilab, http://www-isd.fnal.gov/

[SAMCP] sam_cp, http://d0db.fnal.gov/sam_cp/

[SAMCP-DESIGN] sam_cp Design, http://d0db.fnal.gov/sam/doc/design/sam_cp_initial_design.html

[FCP] Farm Remote Copy utility, http://www-isd.fnal.gov/fcp/pres/FCPPres_frame.htm

[MCFARM]: Monte Carlo Farm at UTA http://www-hep.uta.edu/~mcfarm/mcfarm/main.html

[PPDG-DOC] *D0 Remote Processing,* PPDG Documents April 2003

BIOGRAPHICAL INFORMATION

Sankalp Jain attended the University of Mumbai, Mumbai, India where he received his Bachelors degree in Computer Engineering in July 2001. After working in Mumbai for a brief amount of time, he started his graduate studies at The University of Texas at Arlington, USA in Fall 2002. He received his M.S. in Computer Science and Engineering in 2004.