GRID SCHEDULING VENEER

BY

GUOBIN HAN

J.S.S.

DISSERTATION SUBMITTED IN ~~PART~~ FULFILMENT OF THE REQUIREMENTS FOR THE AWARD OF THE DEGREE OF MASTER OF SCIENCE IN COMPUTING SCIENCE IN THE UNIVERSITY OF GLASGOW.

February 2005

Abstract

In the e-Science community, three local resource management systems are commonly used: Condor, OpenPBS and Sun GridEngine. E-Scientists must determine which system is being used to allow them to select which execution site a job should be dispatched to. If they wish to run their applications on an alternative local resource management system, they must rewrite the job description and submit it in a different way.

The Grid Scheduling Veneer (GSV) is a middleware solution which provides a single job description language to support a subset of features exhibited by Condor, OpenPBS and SGE. The GSV directs a received job to any of these systems available, based upon current load and job attributes. It also provides additional functionality over each local system.

In this dissertation, we discuss the motivation for the GSV project. Subsequently, we describe the methodology and implementation of the system. To test the system, we have designed and deployed a three stage experiment. The experimental results show that the GSV fulfils the requirements and goals of the project.

Table of Contents

Chapter 1 - Introduction

1.1 Why a Grid Scheduling Veneer?

1.1.1 The main reasons for the project

After the pioneering researchers successfully deployed 60 applications on I-Way, people started to realize that Grid Computing can make global computing resource sharing a reality [1]. Along with the user convenience enabled by the Grid, they also discovered some problems when they submitted jobs to the Grid system.

Actually, in the Grid world, a user's job will be processed by a Local Resource Management System (LRMS) or a Batch and scheduling system [2]. There are several LRMS's used in the Grid today. Among them, three are most commonly used – Condor, OpenPBS and Sun Grid Engine (these three systems will be discussed in Chapter 3). When a Grid user or application wants to run a job on a remote system, firstly, they must write a job description informing the LRMS about the elements of the job - i.e. the executable file, arguments to the program, I/O streams and so on. Since each LRMS has its own job description language, which is incompatible with the others, the submitter needs to know which LRMS the job will be processed on and write the corresponding job description that will be understood by the remote system. Thus, the heterogeneity

of the LRMS's introduces unneeded complexity to the submission of jobs. The Grid Scheduling Veneer should understand a single job description language which is the superset of the job descriptions languages of these three LRMS's and translate it into the LRMS-specific description for the LRMS to which the job is finally dispatched.

Another goal of the project is to eliminate the need for a Grid user to know beforehand the execution site to which the job should be sent. When a job is submitted to an LRMS, the user must specify the execution site's address directly when using Globus Toolkit 3 (GT3) [3]. If there are many remote systems, it is annoying to have to decide a job's destination by hand. So the Grid Scheduling Veneer ought to act as a dispatcher which holds information about the available execution sites and allocates a site to each job. Of course, it should also stage in the executables and other files needed and stage out generated files and the output and error streams back to the client.

Finally, the Grid Scheduling Veneer should also dispatch jobs based on static and dynamic information regarding load and other attributes of the LRMS's. In the current situation, for instance, Globus doesn't have a mechanism for routing jobs to the least-loaded execution site; in most situations, it's more optimal to keep the load balanced across the sites in the pool. Each job usually has some constraints or preferred system parameters such as operating system, architecture, memory size. The current GT3 scheduler does not have this function. Therefore, the Grid Scheduling Veneer should keep static and dynamic

system information about the nodes available for use in the LRMS's available to it; it will need to determine this information at startup, and will need to refresh the dynamic information at regular intervals. When a job is received for dispatch, the scheduler will determine where the job should go by using this information.

1.1.2 Overview of the results

In my research over the past 12 months, I have studied three LRMS's (Condor, Sun Grid Engine and Open PBS) and determined the differences concerning scheduling algorithm, acceptable job types, etc. in the first stage. Thereafter, I focused on the Open Grid Services Architecture (OGSA) and the Globus Toolkit 3 (GT3), with the goal of understanding the concept of Grid Services, and how to use Grid Services and GT3 to achieve the project goals. In the last half year, the Grid Scheduling Veneer was designed and implemented. Finally, several experiments were conducted to test whether the goals of the Grid Scheduling Veneer had been achieved and to obtain overall performance data. The system development and experiments have been completed, and most of the system goals have been met.

1.2 Description of each section

"Chapter 1 – Introduction" introduces the project and summarizes the main results.

"Chapter 2 – Background Information" provides background information for the project including the current situation in Grid Computing and development tools used by the project, i.e. the Globus Toolkit.

"Chapter 3 – Three Local Resource Management Systems" discusses the studied LRMS's – Condor, Sun Grid Engine and Open PBS.

"Chapter 4 – Problem Statement" details the questions that the project attempted to answer and why this problem should be studied.

"Chapter 5 – Related work" lists the efforts in recent years that attempted to address the problems discussed in the Chapter 4.

"Chapter 6 – Methodology" describes the approach used to answer the questions raised in Chapter 4. These include translating from the job description language of the Grid Scheduling Veneer to that required for each LRMS, static and dynamic system information gathering, and scheduling a job using the Grid Scheduling Veneer.

"Chapter 7 – System Design and Implementation" discusses the system structure and the implementation.

"Chapter 8 – Results" details the design of experiments to test the Grid Scheduling Veneer and the results obtained from performing these experiments.

"Chapter 9 – Conclusion and future work" states conclusions of the project, summarizes the contributions made, and indicates some issues that need to be addressed in the future.

Chapter 2 - Background Information

This chapter introduces Grid Computing and some other background information related to the project.

2.1 Grid Computing

2.1.1 Brief review of Grid history

Grid Computing has been with us for almost a decade, but the exact definition of Grid Computing is still somewhat ambiguous. Berman and Fox [1] gave the following definition: 'The Grid is the computing and data management infrastructure that will provide the electronic underpinning for a global society in business, government, research, science and entertainment. It integrates networking, communication, computation and information to provide a virtual platform for computation and data management in the same way that the Internet integrates resources to form a virtual platform for information."

Foster and Kesselman [4] defined Grid technologies and infrastructures as "supporting the sharing and coordinated use of diverse resources in dynamic, distributed 'virtual organizations' (VOs)", where "a VO is a set of individuals and/or institutions defined by such sharing rules" [5].

The former definition describes Grid Computing from the point of view of application fields, and the latter gives a logical view of it. Despite these different

points of view, the two definitions agree that resource sharing is at the core of Grid Computing. Grid Computing allows users to securely access remote computing resources through the network.

In the early stages of the Grid's history, it focused on linking a number of supercomputing sites to provide computational resources to a range of high-performance applications. At that time, two representative projects were FAFNER [6] and I-WAY [8]. These two projects had to overcome a number of similar difficulties, including communications, resource management, and the manipulation of remote data, to be able to work efficiently.

Second-generation Grid efforts emphasized middleware to support access to large-scale data and computation resources. Scientists use middleware to hide the heterogeneous nature of the underlying infrastructure, thus providing users and applications with a homogeneous and seamless environment by providing a set of standardised interfaces to a variety of services. In this generation, there are a growing number of Grid-related projects, dealing with areas such as infrastructure, key services, collaborations, specific applications, and domain portals. Among them, Globus [7] and Legion [9] are the most prominent. Globus provides a software infrastructure that enables applications to handle distributed heterogeneous computing resources as a single virtual machine. Legion is an object-based 'metasystem [24]', which provides the software infrastructure so that a system of heterogeneous, geographically distributed, high-performance machines can interact seamlessly. At the same time, several batch and scheduling

systems were developed to manage jobs or tasks within a single domain. These middleware systems formed the base elements of the Grid.

Currently, with the evolution of the Grid Computing infrastructure, a new generation Grid is emerging in which the focus shifts to distributed global collaboration, a service-oriented approach and information layer issues. It supplies a solution for reuse of existing Grid components and information resources, and assembles these components more flexibly. The representation of the third generation Grid is the Open Grid Service Architecture (OGSA) [5]. The OGSA adopts Web Services [41] to offer a service-oriented, autonomic and agent-based solution to Grid Computing. The OGSA will be discussed in detail in section 2.3.

2.1.2 Grid architecture

A Grid system integrates many applications and protocols to provide scalable, secure, high-performance mechanisms for discovery of and negotiating access to remote resources. Standard protocols, which define the content and sequence of message exchanges used to request remote operations, have emerged as an important and essential means of achieving the interoperability that Grid systems depend on. Also essential are standard application programming interfaces (APIs), which define standard interfaces to code libraries and facilitate the construction of Grid components by allowing code components to be reused.

Actually, most of these applications and protocols were not specifically developed for Grid Computing, so it's difficult to draw a clear picture for the Grid architecture. But we can still classify them into four layers. See Figure 1 below [10].

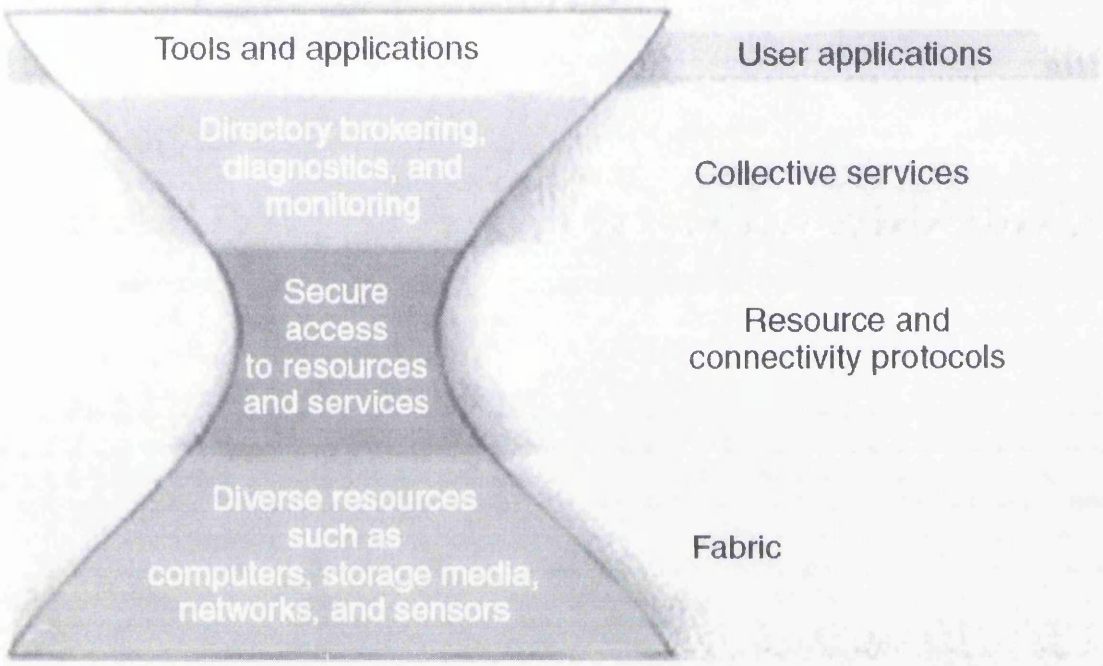

Figure 1

Source: Foster, I. (2002), "The Grid: A new infrastructure for 21st century science", *Physics Today*

At the centre of this hourglass structure are the resource and connectivity layers, which contain a relatively small number of key protocols and application programming interfaces that must be implemented everywhere. The surrounding layers can, in principle, contain any number of components.

At the bottom is the fabric layer, which includes the physical devices or resources that Grid users want to share and access, including computers, storage systems, catalogues, networks, and various forms of sensors.

The layer above is the core layer of the architecture, the connectivity and resource layers. The protocols in this layer must be implemented everywhere, so they should be distilled into a relatively small number. The connectivity sub-layer contains the core communication and authentication protocols required for Grid-specific network transactions. Communication protocols enable the exchange of data between resources, whereas authentication protocols build on communication services to provide cryptographically-secure mechanisms for verifying the identity of users and resources; the resource sub-layer contains protocols that exploit communication and authentication protocols to enable the secure initiation, monitoring, and control of resource-sharing operations. Running the same program on different computer systems depends on resource layer protocols.

The collective services layer contains protocols, services, and APIs that implement interactions across collections of resources. Because they combine and exploit components from the relatively narrower resource and connectivity layers, the components of the collective layer can implement a wide variety of tasks without requiring new resource-layer components.

The topmost layer of any Grid system is the application layer which offers the user access to the Grid through the layers below. Grid applications should have the following functions [10]:

1) Obtain the necessary authentication credentials and verify those credentials when a user tries to access Grid resources.
2) Query information concerning each required resource, i.e. computers, storage systems, and networks, and the location of required input files, to determine if the needed resources are available for the job to be executed.
3) Request the needed resources and submit a job to the system, which will control the staging of the job, initialize the environment, and initiate job execution.
4) Monitor the progress of the various computations and data transfers, notifying the user when all are completed, and detecting and responding to failure conditions

2.2 Globus and Legion

When a computational Grid [4] or a Data Grid [22] needs to be constructed, there are many applicable tools that could be used. Among these tools, the predominant systems are Condor-G [21], Globus and Legion. These systems provide multi-level tools for designing, developing and deploying the production

Grid system. Because Condor-G is based on Globus, here we focus the discussion on Globus and Legion.

2.2.1 Globus

The Globus project has been developed by Argonne National Laboratory and other institutions. Globus provides a software infrastructure that enables the construction of computational Grids.

The primary output of the Globus project is the Globus Toolkit, which is a community-based, open-architecture, open-source set of services and software libraries to support Grid constuction and Grid application implementation.

The toolkit includes software for security, information infrastructure, resource management, data management, communication, fault detection, and portability. It is packaged as a set of components that can be used either independently or together to develop applications.

The core infrastructure of Globus Toolkit 3 (GT3 Core) is based on the Open Grid Services Infrastructure (OGSI) [25] primitives and protocols. The main design goal has been to make the OGSI technology easy to use, reuse, and extend when developing new Grid applications.

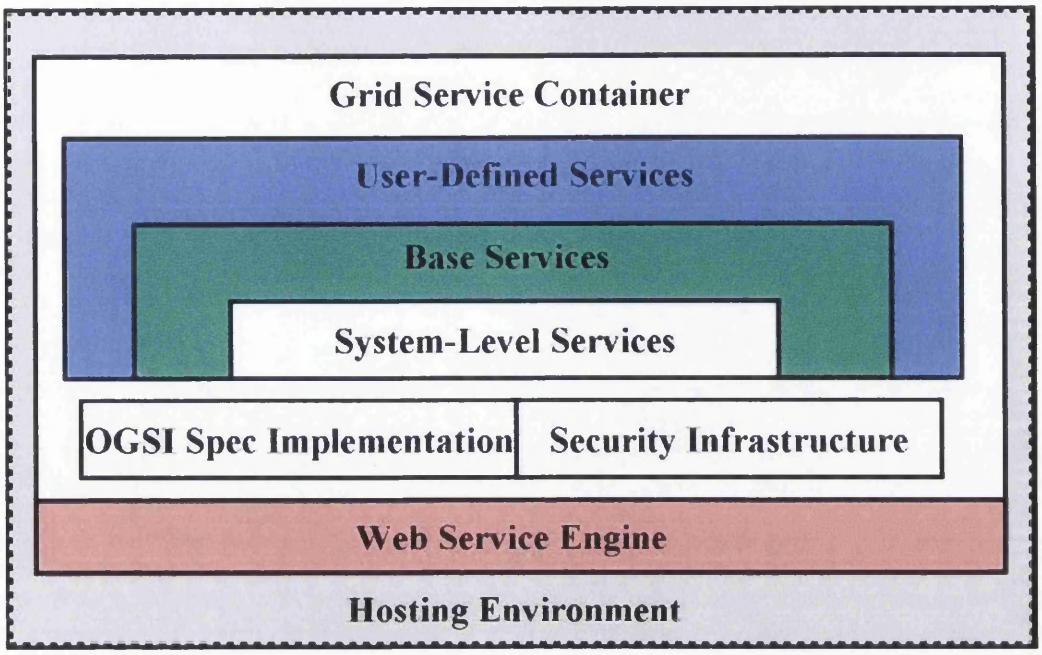

Figure 2

Source: Sandholm, T. Gawor, J. "Globus Toolkit 3 Core – A Grid Service Container Framework"

The components shown in Figure 2 have the following meanings:

- OGSI Spec Implementation: An implementation of the OGSI Specification, i.e., an implementation of all OGSI-specified interfaces.
- Security Infrastructure: This component provides support for message level security, authentication and gridmap-based authorization. The message level security implementation provides session-level (GSI-SecureConversation) as well as message-level (GSI-SecureMessage) security.
- System level services: OGSI-compliant grid services that are generic enough to be used by all other grid services.
- Base services: Not in the GT3 core, but provide some high-level services that a user service can invoke.

The Base services and tools provided by Globus Toolkit 3 include:

- Job management services (JMS): The Job management services provide a way to submit and monitor remote jobs in the Grid. It follows the

interfaces defined in OGSI by using the Web Service Definition Language (WSDL), which is based on XML. Job management services provide a client-side command called managed-job-globusrun that invokes the Master Managed Job Factory Service (MMJFS) to submit a job.

- Index services: Index services are mainly used in discovery operations. Basically, they provide a way to query and produce service data. Index services provide a client-side command called ogsi-find-service-data. The command allows you to query any service data element from any Grid service.
- Grid File Transfer Protocol (GridFTP): This data management tool provides support to transfer files among machines in the Grid and for the management of these transfers.
- RFT services: RFT, also known as multiRFT, is part of the Data Management implementation, along with GridFTP and the Replica Location Service (RLS). It provides the interface for reliable file transfers on Grid servers. A RFT client-side Java-based program is provided (RFTClient).

Globus evolved from the first generation Grid system – I-WAY. The current version of the Globus Toolkit (version 3) provides a set of basic services which follows the Open Grid Service Architecture (detailed below).

2.2.2 Legion

"Legion is a Grid architecture as well as an operational infrastructure under development since 1993 at the University of Virginia" [23]. It is an object-oriented 'metasystem', a collection of disparate resources, usually geographically separated, that can be used as a single resource [24], which provides a single-system view to cover the reality of the underlying, heterogeneous, geographically-distributed systems from which it is composed.

Legion also delivers APIs to access a set of core objects that service the basic metasystem. The following are the core object types of the Legion system.

- Class objects: Each class object manages particular instances of that class. It acts as a manager or policy maker and is in charge of creating new instances, activating and deactivating them and providing client bindings to them.
- Host objects: A host object symbolizes a processing resource in Legion. The resource may represent a single processor or multiprocessor, a mainframe or a collection of several machines.
- Vault objects: Vault objects are responsible for managing objects' persistent representations in Legion. A vault directly accesses the storage device on which a Legion object instance's persistent representation is stored.

- Implementation objects: A Legion implementation object is a program executable, similar to a UNIX file. Such an object cannot be modified once it has been created.
- Implementation cache objects: Each cache object caches implementation objects locally to reduce communication costs.

The first version of Legion was released in November 1997. In 1999, Applied MetaComputing was founded to carry out the technology transition from academia to industry. Then Legion was renamed to Avaki along with the change of company's name in 2001.

2.2.3 Globus vs. Legion

Both Globus and Legion (Avaki) are reasonable options for developing a Grid application (e.g., the Grid Scheduling Veneer) since each of them provide elements that are needed to construct the environment of a Grid system. Moreover, they each include a toolkit supporting application implementation. So, which one is chosen to develop particular Grid system depends on the following attributes of the two tools.

1 Architecture

- The core of Globus is OGSA, which is based on Grid Services (a variant of Web Services). As a result, it inherits all characteristics of Web Services (described below).

- Legion implements a simplified Remote Procedure Call (RPC) style model [42].

2 Applicable scope

- Globus' main focus is to provide fundamental low-level Grid Services for assembling the services to build a Grid system or application. Grid users need to add extra components to form an entire Grid system before the system can be put into use.
- Legion provides a Grid environment for users and programmers by providing higher-level system models. As a result, Legion is a ready-to-use Grid system.

3 Market

- Globus is a community-based, open-architecture, open-source project for academic purpose.
- Legion (Avaki now) calls itself the first commercial Grid product in the world. Therefore, it is difficult to obtain and modify the source code to satisfy specified requirements.

4 Toolkit

- Because Globus is open-source, a programmer can modify the source code to fit into the developing system. Currently, the Globus Toolkit supports the Java and C programming languages.

- Because Legion is an object-oriented system, a programmer can override the core classes' behaviour by modifying and extending these classes.

2.3 Open Grid Services Architecture

There is a significant increase in the number of computational grids in the scientific community. There are at least two architectures that can be used for constructing these grids; use of multiple architectures and toolkits leads to interoperability problems between grids constructed using different architectures. The scientific community has realized this, and has produced the Open Grid Services Architecture (OGSA) as a candidate, standard architecture. This sections describes the OGSA.

2.3.1 Introduction of OGSA

The Open Grid Services Architecture (OGSA) [11] builds upon [5], where an open Grid Architecture is presented, and the technologies and infrastructure of the Grid 'supporting the sharing and coordinated use of diverse resources in dynamic distributed Virtual Organizations (VOs)' [13] are defined.

OGSA brings the Grid and Web Services together to address the problem of services across a distributed, heterogeneous, dynamic, and virtual organization. OGSA extends Web Services to make them more suitable for Grid Computing

environments. For example, Grid services can maintain state for their lifetime; alternatively, Grid service state can be checkpointed and restored.

Grid Services extend Web Services concepts by laying out a set of well-defined interfaces that address discovery, dynamic service creation, lifetime management, notification, manageability, and a set of conventions for naming and upgradeability. These interfaces and conventions are vital for allowing reliable interoperability between services and invoking applications.

Grid Service models have the following properties:

1) Dynamic Service Creation

OGSA uses the concept of a service factory (Factory portType); a service factory is used to create an instance of a desired service. Since the factory is a long-lived service, we can always create another service instance at any point in the future.

2) Dynamic Service Management

OGSA addresses the need for dynamic service management through the provision of the HandleResolver portType, Grid Service References (GSR's), and Grid Service Handles (GSH's). A GSH provides a unique ID for a service instance, and it will not be changed during the entire lifetime of the service. But a GSR is the only way to use the service instance. Once the service instance has been created, a user can convert a GSH into GSR which contains protocol and instance-specific data to allow connection to the service instance.

3) Upgradeability and compatibility

OGSA defines mechanisms for service changes to generate notifications to interested clients. An OGSA client can also retrieve the supported operations of a service before calling the service and the network protocols that can be used to communicate with the service.

4) Lifetime management

OGSA addresses lifetime management by providing "soft-state approach" within the GridService portType. "Soft-state" implies that the client registers interest in a service for a period of time; if the period expires, and no affirmation of interest is made, the service is terminated,

5) Registration/Discovery

OGSA provides "service data" elements within the GridService portType and Registration portType.

The following table summarizes the defined Grid Service portType's:

PortType	Operations	Description
GridService	findServiceData	Query instance service data
	setServiceData	Modify instance service data
	requestTerminationAfter	Soft state management of lifetime
	requestTerminationBefore	Soft state management of lifetime
	Destroy	Explicitly terminate instance
NotificationSource	Subscribe	Subscribe to notifications
NotificationSink	deliverNotification	Receive notification
NotificationSubscription	None	
HandleResolver	findByHandle	Returns one or more references for the given HandleSet (one or more handles)
Factory	createService	Create a Grid Service Instance

ServiceGroup	None	
ServiceGroupRegistration	Add	Create a new entry in a group
	remove	Remove an entries from a group
ServiceGroupEntry	None	

2.3.2 Grid Resource Allocation Management (GRAM) Service

Since the Grid Scheduling Veneer is designed as a bridge between the Grid user and applications/ resources, the GRAM Service is essential to the project. It is shown diagrammatically in Figure 3 below.

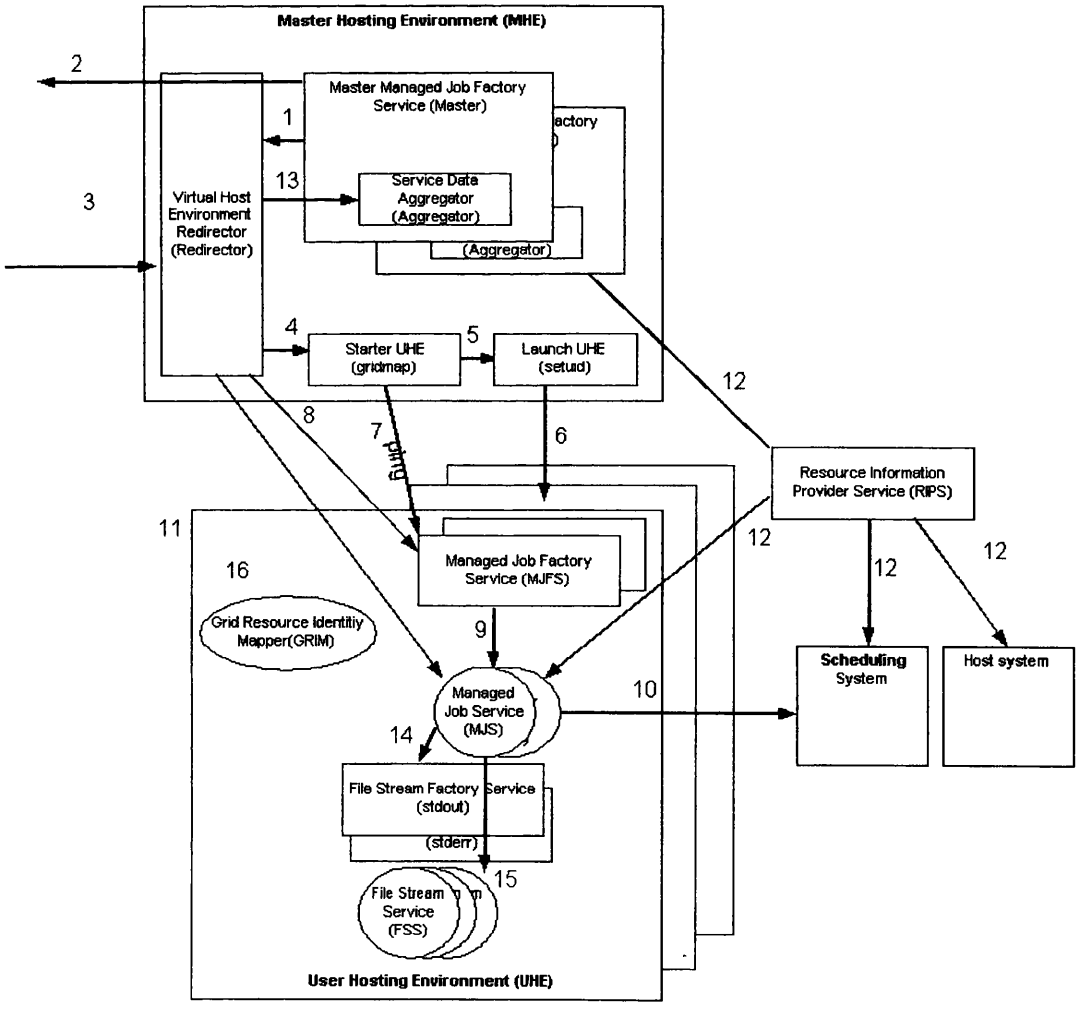

Figure 3

Source: Globus,
http://www-unix.globus.org/toolkit/docs/3.2/gram/ws/developer/architecture.html

1. Virtual Host Environment Redirector

This component accepts all incoming Simple Object Access Protocol (SOAP) [12] messages and redirects them to the appropriate User Host Environment (UHE). This component is part of the Core.

2. Starter UHE

This java class is used by the Redirector to resolve incoming calls to a user hosting environment. The *gridmap* file is used to obtain the username corresponding to a particular subject Distinguished Names (DN), and one UHE is run per user on a machine.

The mapping from a username to the port number of the UHE for that particular user is maintained in a configuration file.

When a request to resolve a URL arrives, and an entry is found in the configuration file, the target URL is constructed and returned to the Redirector. If the UHE on that port number is not active, the setuid/launch module is used to launch a UHE for the user.

If an entry does not exist in the configuration file, a free port number is chosen, the setuid/launch module is used to start up a UHE on that particular port number for the user, and the target URL is returned to the Redirector, after ensuring that the UHE is running. The configuration file is also updated with this entry.

3. LAUNCH UHE

A simple java class that is used to call a C program in order to start a new hosting environment under the user's account. The setuid C program does an su/fork/exec of a shell script that uses a java program to configure and startup the UHE. The C program needs to be "setuid" to "root". The path to the shell script is determined when the C program is compiled. This limits the root exposure to starting up a new hosting environment as a user.

4. Master

The Master Managed Job Factory Service is responsible for exposing the virtual GRAM service to the outside world. It configures the Redirector to direct createService calls sent to it through the Startup UHE, and launches the UHE. The Redirector is instructed to redirect subsequent createService calls that it receives to a user's hosting environment.

The Master uses the Service Data Aggregator to collect and populate local Service Data Elements (SDE) which represent local scheduler data (e.g. freenodes, totalnodes) and general host information (e.g. host cpu type, host OS). If the findServiceData request is for any known Managed Job Factory Service (MJFS) SDE, then it is redirected to the MJFS of the appropriate UHE. All other findServiceData queries are handled locally.

5. Managed Job Factory Service (MJFS)

The Managed Job Factory Service is responsible for instantiating a new Managed Job Service (MJS) when it receives a createService request. The MJFS exists for the life of the UHE.

6. Managed Job Service (MJS)

An OGSI service that, given a job request specification, can submit a job to a local scheduler, monitor its status and send notifications. The MJS will start two File Streaming Factory Service (FSFS) instances, one for the job's stdout and one for the job's stderr. The MJS starts the initial set of File Stream Service (FSS) instances as specified in the job specification. The FSFS's Grid Service Handles (GSH) are available in the SDE of the MJS, which will enable the client to start additional FSS instances of stdout/err or terminate existing FSS instances. The MJS destroys the stdout and stderr File Stream Factories during its preDestroy operation.

7. File Stream Factory Service (FSFS)

The File Stream Factory Service is responsible for instantiating a new File Stream Service instance when it receives a createService request. It exposes two SDE's: the path to the local file being streamed and the current size of the file.

8. File Stream Service (FSS)

An FSS is an OGSI service that, given a destination URL, will stream from its associated local file stream (stdout or stderr) to the destination URL. It exposes two SDE's: the URL of the stream destination and a done flag indicating that the streaming of the file has been completed.

9. Resource Information Provider Service (RIPS)

RIPS is a specialized notification service providing raw data about a resource scheduling system, file system, host system, etc. Some of the data may be privileged. The MJS instances subscribe to RIPS for notification of job state

changes. The Master subscribes for data about the local scheduler (e.g free / total nodes), file system and host system information.

2.3.3 Resource Specification Language (RSL)

Each job management system defines a job description language for specifying a job's requirements and attributes. The OGSA has defined such a language, called the Resource Specification Language (RSL).

The RSL provides a common interchange language to describe resources. The various components of the Globus Resource Management architecture manipulate RSL strings to perform their management functions in cooperation with the other components in the system. The RSL provides the skeletal syntax used to compose complicated resource descriptions, and the various resource management components introduce specific <attribute, value> pairings into this common structure. Each attribute in a resource description serves as a parameter to control the behaviour of one or more components in the resource management system.

2.4 Web Service vs. Grid Service

In OGSA, each component is represented as a Service, regardless of whether it is software or a resource. Each such service is termed a Grid Service, which is a service that is compliant with the Open Grid Services Infrastructure specification, and which exposes itself through a Web Services Description

Language (WSDL) interface. We can view a Grid Service as a standard Web Service but adapted to the requirements typically found in Grid Computing.

The differences between a Grid Service and a Web Service can be summarized as follows:

1. Naming:

   Web Services are addressed with URIs. Since Grid services are Web services, they are also addressed with URIs. However, as defined in OGSI, a "Grid service URI" is called a Grid Service Handle (GSH). In order to meet the requirements to communicate with the service, a GSH must be resolved to a Grid Service Reference (GSR). A GSH points to a Grid Service and a GSR indicates the way to communicate with the corresponding Grid Service.

   Each GSH must be unique and point to a Grid service instance. There cannot be two Grid service instances with the same GSH. A GSH can be thought of as a permanent network pointer to a particular Grid service instance. The GSH does not provide sufficient information to allow a client to access the service, so a client needs to "resolve" a GSH into a GSR. The GSR contains all information that a client requires to communicate with the service.

2. Service data

   Service data is probably one of the most important concepts of Grid services. Service data is a structured collection of information that

is associated with an instance of a Grid service. It is a mechanism to expose a service instance's state data to service requestors. Therefore, these requestors (clients or servers) are able to query, update, and change SDEs that makes up an instance's service data.

Every Grid service instance, by default, has a few standard SDEs, and each one can be of a different type. Grid service instances hold the values of the service data, which can be queried at anytime or be associated with a callback notification when its value changes. For that, OGSI provides an interface to query for SDEs or to subscribe to notifications

3. Notification:

Notifications create a mechanism to allow a notification source to deliver a message to a notification receiver (also known as a notification sink). In GT3, the notification cycle is managed by a subscription management service.

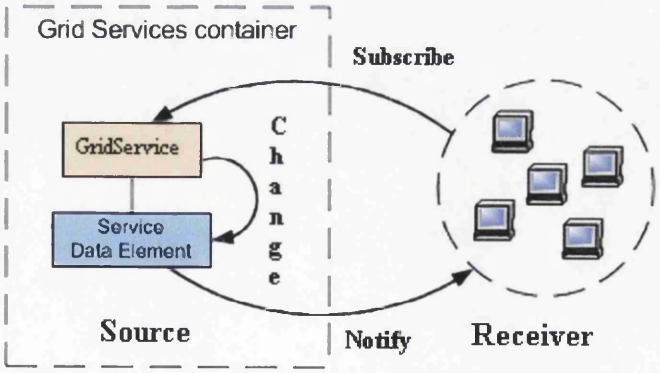

Figure 4

4. Life cycle

The OGSI specification defines the life cycle of any Grid service instance to be "demarcated by the creation and destruction of that service instance" [14]. It is one of the core properties of a Grid service. Actually, the mechanism is part of the hosting environment. A Grid service also supports notifications of lifetime-related events.

Creating a Grid service instance may be done by requesting a Factory to create an instance for the client, while destroying it may be done by invoking a method on the service instance itself. Also, a service instance may be destroyed when the predefined time expires and no reaffirmation is made.

Chapter 3 - Three Local Resource Management Systems

A Local Resource Management System (LRMS) is a system whose functions focus on job scheduling and resources management within a single domain. In Grid systems, an LRMS is the final destination of a job, which means the job obtains its resources from the LRMS and then is executed by the LRMS.

There are several LRMS's available, such as LoadLeveler [26], Maui [27], NQE [28]; a detailed list can be found in [29]. Among them, Condor [15], OpenPBS [16], SGE [17] and LSF [30] are most commonly used.

- Condor: Condor is a distributed system for high-throughput computing developed at the University of Wisconsin [31]. Condor can run on a variety of UNIX platforms, as well as Windows. Besides providing dispatching functionality similar to the other LRMS's, Condor also supports job checkpointing and migration. Condor supports dynamic resource pools that collect idle computers in the management domain.
- OpenPBS: The Portable Batch System (PBS) is "a batch job and computer system resource management package." [32] PBS can run on most UNIX platforms. OpenPBS is the free and open-source version of the PBS product line. It conforms to the POSIX 1003.2d Batch Environment standard.

- SGE: Sun Grid Engine (SGE) manages computational resources based upon policies that govern the automatic gathering, allocation and delivery of resources. SGE allows user to submit batch jobs, interactive jobs and parallel jobs. Checkpointing is also supported by SGE, but this function depends on the platform used.
- LSF: As a Grid product of Platform Computing Corp, the Load Sharing Facility (LSF) is the top commercial job management system in the world. LSF comprises distributed load sharing and batch queuing software that manages, monitors and analyses the resources and workloads on a network of heterogeneous computers, and has fault-tolerance capabilities.

In the Grid Scheduling Veneer project, three Local Resource Management Systems are of interest, i.e. Condor, OpenPBS and SGE. The essential reason to choose these systems is that all of them are free and open-source. This makes easier to study them thoroughly. In contrast, the LSF is a commercial system. It is unlikely that we can view its source code to explore the system structure. Another reason is Condor, OpenPBS and SGE are the most popular systems in Grid research. Many current Grid projects are related to them. For example, Sun Data and Compute Grids [34] is a project based around SGE. Condor on WAN [35] is a project related to Condor. As a result, significant amounts of useful information about these systems are available in the research community.

In the following sections, we compare the system structures and scheduling algorithms of the three systems.

3.1 Condor

3.1.1 System structure

Four types of machine – central manager, checkpoint server, submitter's machine and job executor – exist in a Condor system. The following graph illustrates the relationship among these machines.

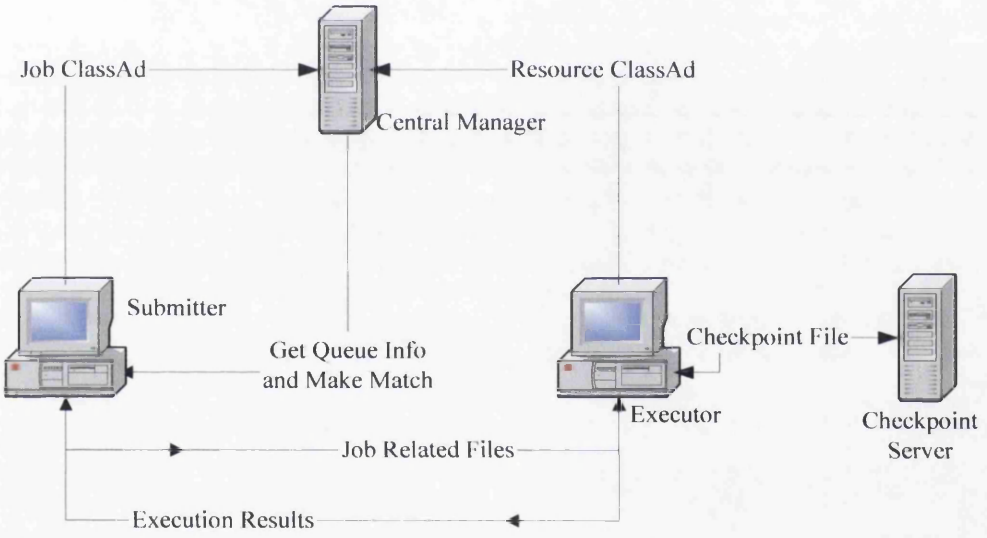

Figure 5

1. Central Manager

This machine is the collector of information, and the negotiator to perform matchmaking of resources and resource requests.

2. Checkpoint Server

It services requests to store and retrieve checkpoint files.

3. Submitter

A user writes a ClassAd to describe the resource requirements and the execution files for a job, and submits the ClassAd to the Central Manager. The Submitter also transfers any job related files to the Executor machine and transfers back any output files.

4. Executor

Runs jobs that match its resource specification, and returns any execution results to the submitter's machine. If the Executor's owner returns to use the machine again, any running, Condor-scheduled jobs are checkpointed.

3.1.2 Software structure

All of the above functions depend on support from daemon processes in the system. In Condor, there are seven daemon processes. Each of the four types of machine has different daemon processes running on them.

1. Central Manager

- Condor_ master

This daemon is responsible for spawning the other daemons and keeping all the rest of the Condor daemons running on each machine in the pool.

- Condor_ negotiator

This daemon is responsible for all the match-making within the Condor system. Periodically, the negotiator begins a negotiation cycle, where it queries the collector for the current state of all the resources in the pool. It contacts each Condor_schedd (details below) that has been waiting for resource requests in priority order, and tries to match available resources with those requests.

- Condor_ collector

This daemon is responsible for collecting all information about the status of a Condor pool. All other daemons (except the negotiator) periodically send ClassAd updates to the collector. These ClassAds contain all the information about the state of the daemons, the resources they represent or resource requests in the pool.

2. Submitter

- Condor_ master

Same as that for the Central Manager.

- Condor_ schedd

This daemon represents resource requests to the Condor pool. Any machine to which jobs may be submitted needs to have a condor_ schedd running. When users submit jobs, the jobs go to the schedd, where they are stored in the job queue, which the schedd manages.

Condor_schedd advertises the number of waiting jobs in its job queue and is responsible for claiming available resources to serve those requests. Once a schedd has been matched with a given resource, the schedd spawns a condor_ shadow (described below) to serve that particular request.

- Condor_ shadow

This program runs on the machine to which a given request was submitted and acts as the resource manager for the request. Jobs that are linked for Condor's Standard Universe, which perform remote system calls, do so via the condor_ shadow. Any system call performed on the remote executor machine is sent over the network back to the condor_ shadow, which then performs the system call (such as file I/O) on the submitter's machine, and the result is sent back to the remote job over the network. In addition, the shadow is responsible for making decisions about the request (such as where checkpoint files should be stored, how certain files should be accessed, etc).

3. Executor

- Condor_ master

Same as that for the Central Manager.

- Condor_ startd

This daemon represents a computational resource to the Condor pool. It advertises certain attributes about that resource that are used to match it with

pending resource requests. When the startd is ready to execute a Condor job, it spawns the condor_ starter, described below.

- Condor_ starter

This program is the entity that actually spawns a remote Condor job on a given machine. It sets up the execution environment and monitors the job once it is running. When a job completes, the starter notices this, sends back any status information to the submitting machine, and exits.

3.1.3 Scheduling in Condor

There are two levels of scheduling mechanism in a Condor system. One is implemented in the submitter's local queue to decide the execution order of jobs, and the other is at the system level related to which machine a job is assigned to for execution.

1. Local queue

The Condor_sched daemon is in charge of the local queue on the submitter's side. Every job is assigned a priority when it is submitted by the condor_submit command. Job priorities assigned to Condor jobs are used to control the order of execution. Such priorities only affect the local queue, which orders jobs by job priority in the condor_schedd daemon.

When a new negotiation cycle occurs, the negotiator chooses the highest priority job – the lowest priority number – in the local queue and makes a match with available resources. A user can use the condor_prio command to change the priority of a job in the local queue.

2. System level

At this level, Condor uses two priorities to determine the next submitter whose job will be selected to run.

- Real User Priority (RUP)

A user's RUP measures the resource usage of the user through time. Every user begins with the default value 0.5 (the highest priority). When a user is

allocated resources to run a job, then the RUP will be increased by the number of resources it obtained.

The RUP will be halved after a certain time (predefined as PRIORITY_HALFLIFE, a time period defined in seconds).

- Effective User Priority (EUP)

The effective user priority (EUP) of a user is used to determine how many resources that user may receive. The EUP is linearly related to the RUP by a priority boost factor, which may be defined on a per-user basis. The default value is 1.0, which means that the EUP is same as the RUP.

The negotiator uses the EUP to determine which user should receive resources. The higher the priority of a user, the more resource it receives.

- Example

The following diagram uses an example to depict the scheduling mechanism in Condor.

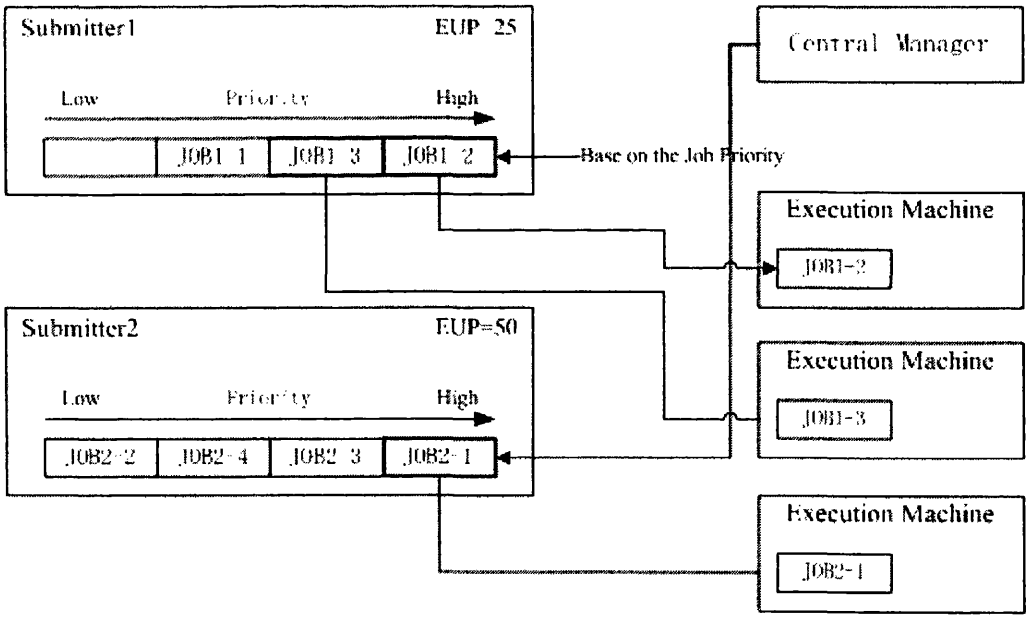

Figure 6

The queue on the submitter's machine is ordered by the priority of the job that the submitter specified. The negotiator on the central manager chooses the next job to schedule from the submitter's machine that has the highest EUP.

Because submitter1's EUP is 25, which is smaller than that of submitter2, JOB1-2 and JOB1-3 were chosen. The number of resources that a submitter may receive is related to the ratio between the EUPs of submitting users inversely. Therefore submitter1 with EUP=25 will receive twice as many resources as submitter2 with EUP=50.

3.2 OpenPBS

3.2.1 System and software structure

Four components – Commands, the job Server, the job executor, and the job Scheduler – work together to form a full OpenPBS system.

Commands

"Commands" is a tool that has two forms: a graphical user interface and a command line interface that conforms to POSIX 1003.2d. A user can submit, monitor, modify, and delete jobs through either version of the tool. Three types of commands exist in OpenPBS: user commands for any authorized user, operator commands and manager (or administrator) commands. Only a user who has enough privilege can access operator and manager commands.

Job Server

The Job Server is the central server which runs the pbs_server daemon. It provides the core functions including receiving/creating a batch job, modifying the job, protecting the job against system crashes, and running the job (placing it into execution).

Job Executor

The job executor is the machine where the job actually runs. The daemon running on the Job Executor, pbs_mom, is the parent of all executing jobs. The job server transfers a job to a job executor and its pbs_mom places the job into execution.

The pbs_mom creates a new session as identical to a user login session as is possible. For example, if the user's login shell is csh, then pbs_mom creates a session in which login is run as well as .cshrc. The pbs_mom also has the responsibility for returning the job's output to the user when directed to do so by the Job Server.

Job Scheduler

The Job Scheduler decides which job should run and where and when it is run based upon the scheduling policy. Pbs_sched is the daemon that is in charge of scheduling in PBS. PBS allows an administrator to customize the scheduler. In this way, the job scheduler presents the scheduling policy of each PBS system. The Scheduler collects the executors' state and communicates with the Server to learn about the availability of jobs to execute. The interface to the Job Server is through the same API as the commands. In fact, the Scheduler just appears as a batch Manager to the Job Server.

3.2.2 Job Scheduler

Server Interaction

When the server (pbs_server) initiates the scheduling cycle, it opens a connection to the scheduler (pbs_sched) and sends a command indicating the reason for initiating the scheduling cycle. Once the server has contacted the scheduler and sent the reason for the contact, the scheduler then becomes a privileged client of the server. As such, it may command the server to perform any action allowed to a manager.

The figure below illustrates the interaction between the server and the scheduler within a single-host environment.

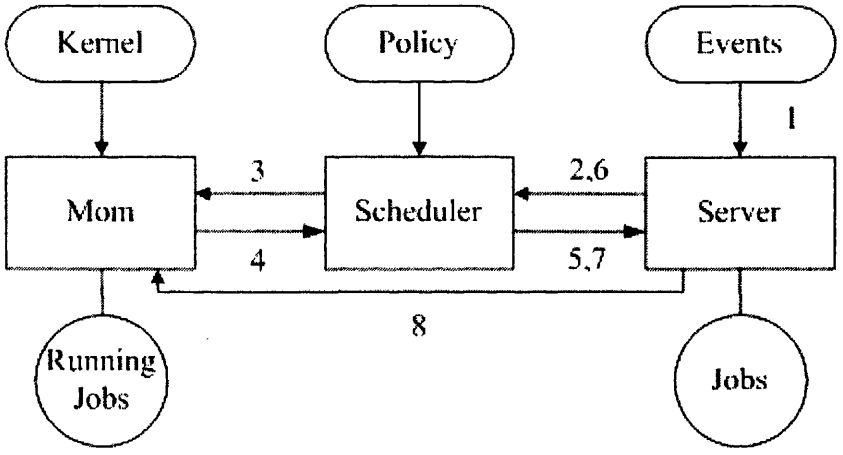

Figure 7

1. Event informs Server to initiate a scheduling cycle
2. Server sends scheduling command to Scheduler
3. Scheduler requests resource information from Mom
4. Mom returns request information
5. Scheduler request job information from Server
6. Server sends job status information to Scheduler and Scheduler makes policy decision to run job

7. Scheduler sends run request to Server

8. Server sends job to Mom to run

Default Scheduler

We describe the "FIFO" Scheduler as the default scheduler, as all the others depend on particular kinds of system structure, such as IBM_SP, SGI and CRAY.

This Scheduler provides several simple scheduling policies. It provides the ability to sort jobs in several different ways, in addition to FIFO order. There is also the ability to sort on user and group priority. Mainly, this Scheduler is intended to be a template for writing a custom Scheduler.

The following are polices used in the FIFO scheduler

- All jobs in a queue will be considered for execution before the next queue is examined.
- The queues are sorted by queue priority.
- The jobs within each queue are sorted by requested cpu time (cput). The shortest job is dispatched first.
- Jobs which have been queued for more than a day are classified as "starving", and heroic measures will be taken to attempt to run them.

Custom Schedulers

A unique feature of PBS is its external Scheduler module. This allows a site to implement any policy of its choice using provided APIs. To provide more freedom in implementing policy, PBS provides three scheduler frameworks.

Schedulers may be developed in the C language, the Tcl scripting language, or PBS's very own C language extensions, the Batch Scheduling Language, or BaSL.

PBS Scheduler is put into a separate process to provide a flexible mechanism by which you may implement a very wide variety of policies. The Scheduler uses the standard PBS API to communicate with the Server and an additional API to communicate with the PBS resource monitor, pbs_mom.

3.3 SGE

3.3.1 System structure

In the Sun Grid Engine system, four types of hosts are involved.

Master Host

Master host is the centre of the SGE cluster. Two daemons run on the master host: One is the master daemon – sge_master; another is the scheduler daemon – sge_schedd. The other Sun Grid Engine components, such as queues and jobs, are under control of the two daemons. The master host also stores status tables of the components, about user access permissions, and the like.

By default, the master host is also an administration host and submit host.

Execution Host

Execution hosts run the Sun Grid Engine execution daemon, sge_execd. There is an execution host list in the Sun Grid Engine system to store the

authorized nodes on which jobs can be executed. Each execution host has a qualified job queue.

Administration Host

A user manages the Sun Grid Engine system on an administration host. The SGE provides a set of command-line interface tools and a graphical interface tool (QMON) to administrate the system.

Submit Host

Submit hosts are allowed to submit and control batch jobs, only. In particular, a user who is logged into a submit host can submit jobs via qsub, and can monitor a job's status via qstat.

3.3.2 Software structure

1. Daemons

Four daemons provide the functionality of the Sun Grid Engine system.

sge_qmaster

Sge_qmaster is the master daemon of the other daemons. It controls the cluster's management and scheduling activities. Sge_qmaster maintains tables about hosts, queues, jobs, system load, and user permissions. It also gathers the current status of nodes in the cluster and sends them to the scheduler. Whenever the scheduler has made scheduling decisions, it informs sge_master. Then the sge_master requests actions from sge_execd on the appropriate execution host.

sge_schedd

The scheduling daemon, sge_schedd, maintains an up-to-date view of the cluster's status. It decides which jobs are dispatched to which queues and then forwards these decisions to sge_qmaster, which initiates the required actions.

sge_execd

The execution daemon is responsible for the queues on its host and for the execution of jobs in these queues. Periodically, it forwards information such as job status or load on its host to sge_qmaster.

Sge_commd

The communication daemon, sge_commd, must run on every host in the cluster. It is used for all communication among Sun Grid Engine components. The communication is based on the TCP protocol.

2. Queues

A Sun Grid Engine queue is a container for a class of jobs allowed to execute on a particular host concurrently. A queue determines certain job attributes; for example, whether it may be migrated. Throughout their lifetimes, running jobs are associated with their queue. Association with a queue affects some of the things that can happen to a job. For example, if a queue is suspended, all the jobs associated with that queue are also suspended.

In the Sun Grid Engine system, there is no need to submit jobs directly to a queue. Users only need to specify the requirement description of the job (e.g., memory, operating system, available software, etc.) and Sun Grid Engine software will automatically dispatch the job to a suitable queue on a low-loaded

host. If a job is submitted to a particular queue, the job will be bound to this queue and to its host, and thus, Sun Grid Engine daemons will be unable to select a lower-loaded or better-suited device.

3.3.3 Scheduling of Sun Grid Engine Jobs

Essentially, a Sun Grid Engine system uses two sets of criteria to schedule jobs:

1. Job Priorities

When determining the order of scheduling precedence of different jobs, a first-in-first-out (FIFO) rule is applied by default. All pending (not yet scheduled) jobs are inserted into a list, with the first submitted job being at the head of the list, followed by the second submitted job, and so on. The Sun Grid Engine software will try to schedule all jobs in the list from head to tail, regardless of whether the previous job has been dispatched or not.

The FIFO order may be overruled by the cluster administrator via priority values being assigned to jobs. The priority value assigned to a job can be positive and negative integers, while the default value at submit time is 0. The pending jobs list is sorted correspondingly in the order of descending priority values. By assigning a relatively high priority value to a job, the job is moved toward the head of the pending jobs list.

2. Equal-Share-Scheduling

The FIFO rule sometimes has a few disadvantages, especially if a user tends to submit a series of jobs at approximately the same time (e.g., via a shell-script issuing one submit request after another). All jobs submitted afterwards and designated to the same group of queues will have to wait a very long time. If the cluster uses equal-share-scheduling, the jobs belonging to users who already have a running job on the system will be queued to the end of the precedence list. The sorting is performed only among jobs within the same priority

3. Queue Selection

The Sun Grid Engine system does not dispatch jobs requesting non-specific queues unless they can be started immediately. Such jobs will be marked as spooled at the sge_qmaster, which will try to re-schedule them from time to time. Thus, such jobs are dispatched to the next suitable queue that becomes available.

Jobs that are targeted at a particular named queue go directly to that queue, regardless of whether they can be started or they have to be spooled. Jobs submitted with non-specific requests use the spooling mechanism of sge_qmaster for queuing, thus utilizing a more abstract and flexible queuing concept.

If a job is scheduled and multiple free queues meet its resource requests, the job is usually dispatched to the queue (among those suitable) belonging to the least-loaded host. By setting the Sun Grid Engine scheduler configuration entry queue_sort_method to seq_no, the cluster administration may change this load-dependent scheme into a fixed-order algorithm: the queue configuration

entry seq_no is used to define precedence among the queues assigning the highest priority to the queue with the lowest sequence number.

Chapter 4 - Problem statement

The Grid Scheduling Veneer project's goal is to improve the ability for a Grid user to access Grid Computing resources. It includes providing a unified job description language to define a job's attributes, forming an execution site pool to hide the complexity of job submission, and making dispatching decisions based on the execution sites' current operational status.

Each sub-goal above addresses one problem that is critical to the client-side use of Grid Computing. In particular: each LRMS supplies a different job description language; a user needs to be aware of an execution site's identity before submitting a job; the site's current status is unknown to the user; some required job attributes are unavailable in one or more of the target LRMS's.

4.1 Different job description languages

For the three LRMS's studied, there exist two kinds of job description languages to convey to the execution site the attributes of a job – ClassAds for Condor and shell scripts for OpenPBS and Sun Grid Engine. Although OpenPBS and SGE both adopt Shell scripts to represent a job's attributes, there are substantial differences between the structures of those scripts.

4.1.1 ClassAds

"Classified Advertisements (ClassAds) are the *lingua franca* of Condor" [15]. In the Condor system, ClassAds are used to describe a job's attributes and requirements. In addition to describing jobs, ClassAds also are also used to communicate with other Condor daemons to exchange information.

A ClassAd provides a mapping from attribute names to expressions. Sometimes, a ClassAd is simply a property list. In this case, the expressions are simple constants, such as integer, floating point, or string. Attribute expressions can also be more complicated. In Condor, an attribute expression can make a comparison with another one. For example, the expression "other.size > 3" in one ClassAd evaluates to true if the other ClassAd has an attribute named size and the value of that attribute is (or evaluates to) an integer greater than three. Two ClassAds match if each ClassAd has an attribute requirement that evaluates to true in the context of the other ClassAd. ClassAd matching is used by the Condor central manager to determine the compatibility of a job and workstations upon which the job may be run.

ClassAds are very powerful. Because the job description is interpreted by the Condor scheduler daemon before being executed directly by a condor starter daemon, it includes hundreds of job expressions to describe the job's attributes, such as executable file and input/output/error streams. It also provides expressions to control a job's lifecycle, for example, files to be staged in/out, environments variables, and scratch directory.

4.1.2 Shell scripts

OpenPBS and SGE don't have their own job description language but use UNIX shell scripts plus some patterns to add control attributes to the scripts. Because the execution daemon will run the script directly in OpenPBS and SGE, a user can easily control the execution of a job. So there are fewer control attributes in these two systems. If a job needs to build an environment to execute the binary, the submitter must explicitly do it in the script. For instance, if a job must read and write some data files in the local directory, the script of that job should explicitly include the file transfer commands to stage in and stage out the files.

Using Shell scripts as a job description language is very flexible for one who has experience with the UNIX operating system, but it is somewhat difficult for the average Grid Computing user. In addition, Shell scripts in different variants of UNIX may not be the same. This makes it more difficult to write a job description using Shell scripts.

The scripts used in OpenPBS and SGE are incompatible. For example, the token of the job attribute in OpenPBS is "#$PBS" and the one in SGE is "#$". Additionally, the expressions for job attributes in the two systems are also different.

4.1.3 Job attributes in three systems

The following table (figure 8) gives a summary of the job attributes commonly used in these description languages.

Type	Name	Description	Condor	SGE	PBS
I/O related	directory	Specifies the path of the directory	initialdir	set in script	set in script
	executable	The name of the executable file to run	executalbe	filename	filename
	stdin	The name of the file to be used as standard in/out/err for the executable on the remote machine.	input	set in script	set in script
	stdout		output	#$o	#PBS -o
	stderr		error	#$e	#PBS -e
	file_stage_in	files to be staged to the nodes which will run the job	fetch_files	N/A	N/A
	append_files	append the output to files	append_files	N/A	N/A
	file_stage_out	files to be staged from the job to a GASS-compatible file server.	Automatically	N/A	N/A
Job Related	environment	The environment variables that will be defined for the executable	environment	#$v	#PBS -v
	arguments	The command line arguments for the executable.	arguments	set in script	set in script
	priority	the priority of the job	priority	#$p	#PBS -p
	checkpoint	Whether the job could be checkpointed	Universal= Standard	#$c	#PBS -c
	date	the time the job to run	N/A	#$a	#PBS -a
	interactive	submit a interactive job	N/A	call qrsh	#PBS -I

Type	Name	Description	Condor	SGE	PBS
Job Related	batchjob	submit a interdependent job batch	call condor_submit_dag	#$hold_jid	N/A
	allow_startup_script	excute a script other than exe (for condor)	allow_startup_script	N/A	N/A
	maxMemory	Specify the max amount of memory required for this job	Memory	h_data	pmem
	disk	Free space in kbytes	disk	h_fsize	file
	maxWallTime	Explicitly set the maximum walltime for a single execution of the executable.	N/A	h_cpu	pcput
System Related	hostCount	Defines the number of nodes	CPUs	N/A	nodes
	arch	Machine architecture	arch	N/A	arch
	kflops	floating point performance as determined via a Linpack benchmark.	kflops	N/A	N/A
	mips	Integer performance	mips	N/A	N/A
	image_size	maximum virtual image size	image_size	N/A	N/A
	coresize	maximum core file size(kb)	coresize	N/A	N/A
Other	Maillist	the list of users to send mail	notify_user	#$M	#PBS -M
	Mail	Under what situation should send mail	notification	#$m	#$m

Figure 8

4.1.4 The problem

From the above discussion, it should be apparent that it is difficult for a Grid user to submit the same job to these three systems. After a job is successfully executed on Condor, the user must rewrite the job description if it will also be

submitted to the other systems. This is especially difficult when the user doesn't know how to write a script on UNIX.

4.2 Complexity

As the number of computational Grids increases, more and more resources become available to the Grid user. But along with the increase in the number of resources, the complexity of job submission also increases.

In OGSA, the Job Managed Service is a proxy for an LRMS on the execution site. When attempting to access the JMS, the caller must provide its GSH, which includes three parts – address, port and service name of the Job Manager Service. See the figure below.

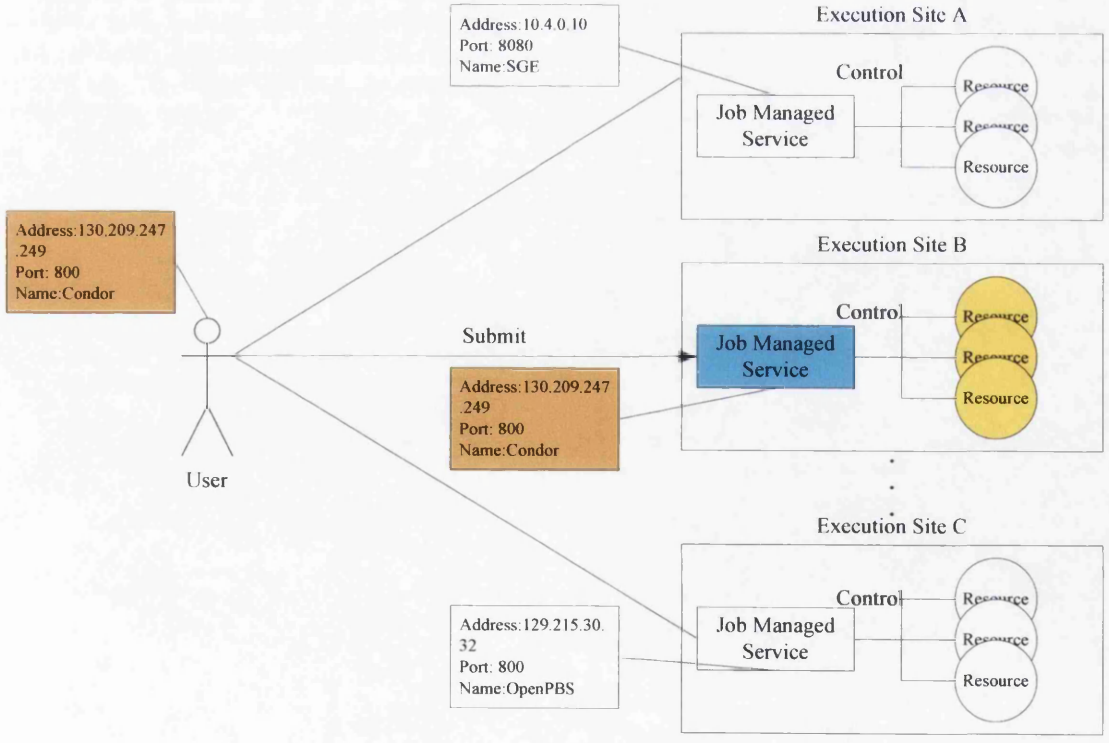

Figure 9

In this scenario, one user has been authorized to access resources in three execution sites. A job is going to be submitted to execution site B. Firstly, the user needs to know site B's basic information, a.k.a. Address, port and service name. Then a user needs to invoke a submission method on one of the client tools in GT3 with this basic information, as parameters. Such an approach is possible if there are only few execution sites available. When the number of available sites increases to a large number, the ability to remember all of this site information turns out to be increasingly difficult.

So, a Grid user needs a pooling system to hold all execution information. When submitting a job, the system should find a qualified site upon which to run the job.

4.3 Blind submission

From section 4.2, we can also deduce another problem that the user will encounter when working on multi-domain Grid systems: because the current API which OGSA offers requires the user to specify the GSH of the Job Manager Service, the user must decide where the job should be executed. Unfortunately, it is not always possible for a user to obtain enough, up-to-date, system information about each execution site, especially when the number of the sites is large. And it's difficult to determine the destination of the job if the user needs to manually consider many attributes of the execution site, such as architecture of the computer, operating system, and queue size.

Imagine the situation in Figure 10. There are dozens of execution sites that the user can use. Where the job goes depends on the user himself, for the OGSA has not supplied a mechanism for obtaining static system information and dynamic information like queue size, system load and so on. The Grid user will submit the job to a few execution sites because it is simpler to remember basic information about a few execution sites. Those execution sites will be busier than others which are ignored due to the non-exhaustive selection algorithm used by the user.

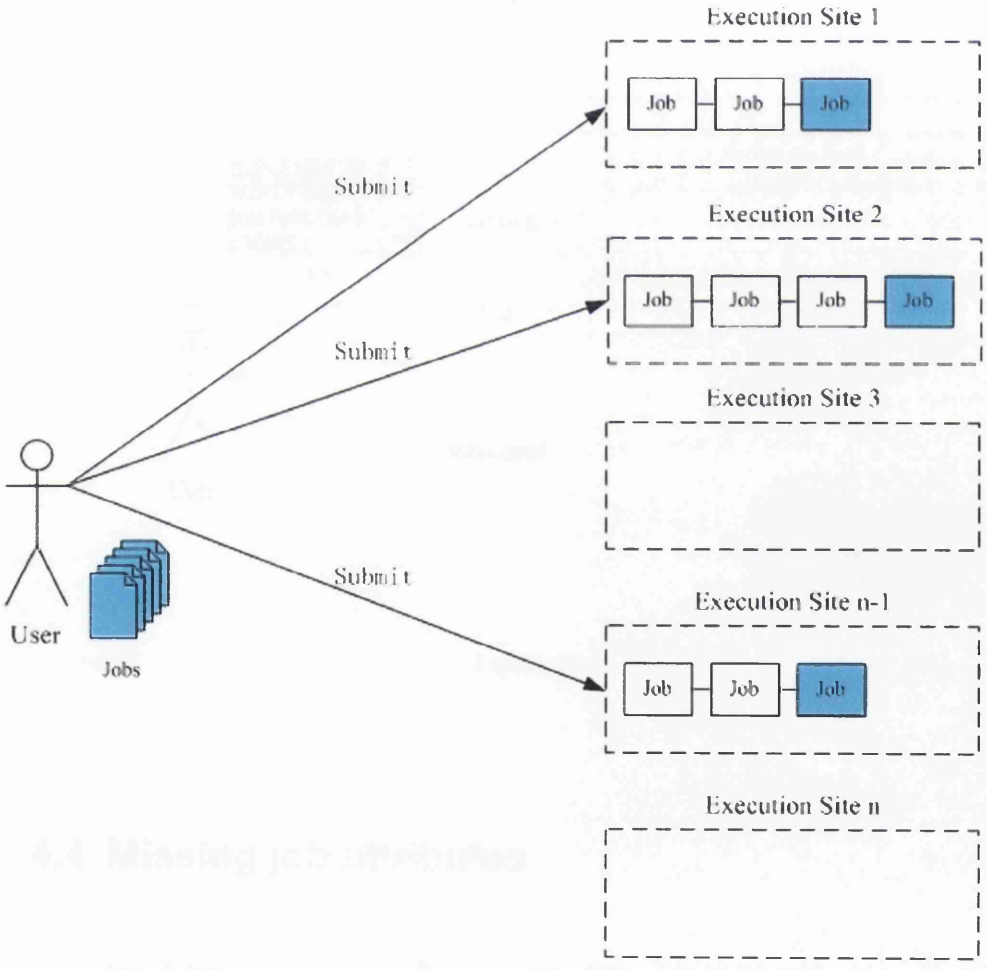

Figure 10

Another scenario (See Figure 11) is that the user submits a job to one site, but it does not satisfy the requirements of the job, for example the runtime size of the executable exceeds the disk size of the execution site. Although the execution will fail, the job will still be accepted. After the environment preparation and stage in processes, the job starts to run on the site. But it fails when the disk space is exhausted, and the site returns an error indication to the user. Unless the user is aware of the exact cause of the failure, he/she will keep trying until the job has successfully finished. The process is very time-consuming and generates unnecessary system load on the execution sites.

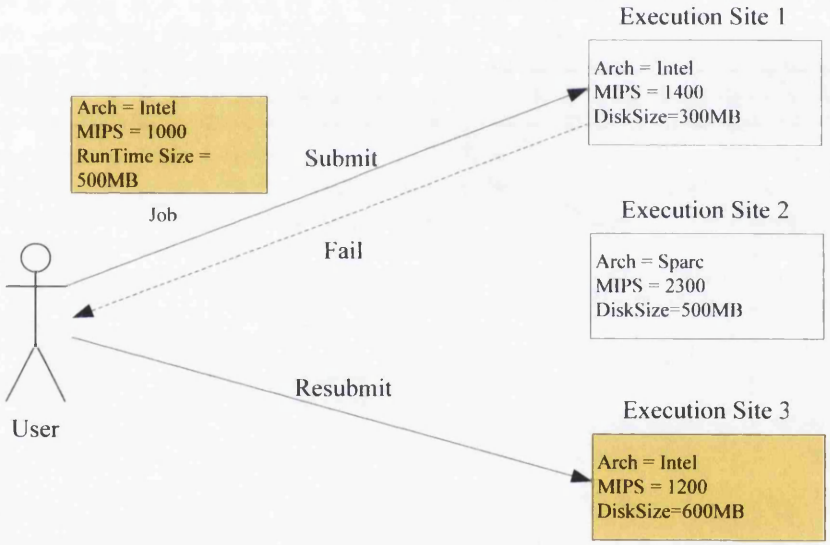

Figure 11

4.4 Missing job attributes

From Figure 8, it is not hard to discover that the number of supported attributes for our three LRMS's are not equal. For instance, Architecture, MIPS

and FLOPS can be found in Condor but not in OpenPBS and SGE. In contrast, Max Walltime is a quite useful attribute in OpenPBS and SGE to terminate the job after a certain time, but it is not available in Condor.

When a job is submitted to a site for which the LRMS on the site cannot accept all the attributes included in the job's description, a job with attributes which are unknown to the LRMS but are critical to the job execution will fail, since unknown attributes will be discarded by the LRMS.

4.5 Brief summary

From the problems listed above, the requirements of the Grid Scheduling Veneer can be defined.

First of all, the Grid Scheduling Veneer should supply a single job description language which contains most of the job attributes of the three LRMS's considered – Condor, OpenPBS and SGE. The job description will be parsed by the Grid Scheduling Veneer and transformed to an appropriate, LRMS-understandable description when the job is submitted to the chosen LRMS.

Moreover, the Grid Scheduling Veneer should bridge the differences amongst the attributes of the three LRMS. More precisely, attributes such as system requirements attributes – Flops, Mips and Architecture etc. – will be pre-processed before the job runs on a particular LRMS.

Thirdly, the Grid Scheduling Veneer should form a queue to hold all the jobs submitted by user and dispatch them based on the job requirements and the destination system information. It demands that the Grid Scheduling Veneer obtain system information about execution sites in the pool and refreshes that information frequently to make sure they are up-to-date.

Finally , although the GSV focus on the functionality rather than the performance, the overhead of the GSV should be within a reasonable scope and the each job's processing time in the GSV should not be related to the job number of one job batch.

4.6 Reasons for choosing GT3

After discussing the requirements of the project, now it is time to decide which toolkit should be used in the GSV – Globus Toolkit or Legion.

From the project's motivation, the tools used to implement the Grid Scheduling Veneer need to have the following properties:

2) Adaptive: Since the Grid Scheduling Veneer will interact with three different LRMS's, the tools must have the functionality or the mechanisms needed by a developer to communicate with those systems.

3) Extensibility: This is very important for developing a Grid application, because it is impossible and unnecessary to implement the system from the very beginning. So the chosen tools should

provide a programming interface and also some reusable elements in order to construct a system easily.

4) Standard: Standard means the adopted tool must follow the commonly-accepted system architecture. Thus, it will reduce future system migration and maintenance costs.

5) Security: Authentication, authorization, and policy are among the most challenging issues in Grids. The Grid Scheduling Veneer should make ensure that the user is authenticated before accessing the system and that all operations are authorized. Because it acts as a dispatcher, it must delegate to the user who submits the job to guarantee appropriate access to resources in the Grid system.

6) Secure and reliable file transfer: Because a Grid user's job will cross multiple domains, the tools should supply a stable and safe file transfer mechanism.

7) Flexible information management system: Information about each LRMS will be gathered by the Grid Scheduling Veneer and stored in a centralized or distributed manner. The API of the tools should remain consistent if the model of the information changes.

8) Extensive support: The tools should be accepted by developers and users in Grid environments to reduce the complexity of development and operability.

The following table makes a comparison between the two candidates for this project – Globus, Legion.

	Globus	**Legion (Avaki)**
Adaptive	The Managed Job Service (MJS) provides a programming interface which can call external java classes or Perl scripts to communicate with the LRMS's.	The Host objects of Legion can federate the most existing schedulers such as SGE and PBS.
Extensibility	Developer can modify the source code of the core service or write a Web Service to call the API provided.	The Legion has a simple, generic default Scheduler. Developer can implement the application-specific scheduler by filling in a bunch of object event functions.
Standard	OGSA, Grid Service	N/A
Security	RSA public key authentication, X.509 certificate, ACL for authorization. OpenSSL for encryption and MD5 for integration	RSA public key authentication, Non-X.509 certificate, ACL for authorization. OpenSSL for encryption and MD5 for integration
File transfer	GridFTP and GASS	A legion-aware NFS server
Information management architecture	MDS3 is based on the LDAP protocol. It can generate, register, index, aggregate,	N/A

	subscribe, monitor, query, or display information freely.	
Extensive support	The Globus is the most popular toolkit used in Grid systems [7][38]. Many ongoing Grid projects are based on Globus, such as OGSA-DAI [36], OGSA-Grid [37], etc.	There are some projects related to Legion, for instance "A Legion CCA Project" [39]. But after Legion was commercialized, few projects now use it.

From the above table, both Globus and Legion address the Adaptive, Security and File transfer requirements. Globus is better for the other attributes, especially Standard and Extensive support. These two attributes are very important for the GSV project. Adopting the industry standard will maintain the continuity of the project, and we can also make good use of other projects' valuable experience. Therefore, we choose Globus as the toolkit for the project.

Chapter 5 - Related work

The problems mentioned in the previous chapter were emerging along with growth of the Grid community and exists for several years. There were some related works to address these problems in recent years, for instance the Sun Data and Compute Grids Project of the EPCC and the Condor-G from Condor project in the University of Wisconsin.

5.1 The Sun Data and Compute Grid (SCG) Project

SCG is a project undertook by the EPCC in University of Edinburgh. "The aim of the project is to develop an industry-strength fully Globus-enabled compute and data scheduler based around Grid Engine, Globus and a variety of data technologies." [34] In this project, the developer of the project tried to solve the problems of the "Complexity" and "Blind Submission" (discussed in the Chapter 4) by implementing a hierarchical scheduler to manage the Grid jobs and transfer the needed files and information to or from the remote sites. The architecture of the scheduler is shown in the following chart.

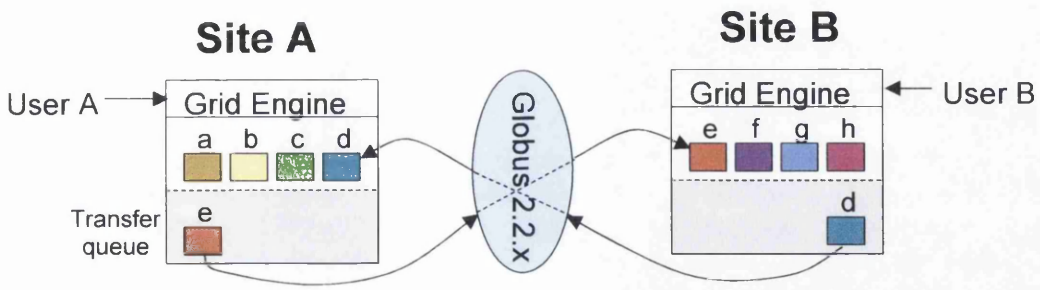

Figure 12

The local scheduler is an integration of SUN Grid Engine job queue and Globus. The SUN Grid Engine job queue buffers the jobs submitted by the local user and the Globus is in charge of the information propagation and files transfer. The local scheduler senses other site's current load information. The job will be transferred to the relatively lightly loaded site when the local site reaches the load limit.

The SCG project partially solves the problems mentioned in Chapter 4. But because the SCG project's scope only is within in the SUN Grid Engine which is the only one of the commonly used LRMS in the Grid world, the SCG limits the user's choice to access other computing resources that are controlled by other LRMS, such as Condor and OpenPBS.

5.2 The Condor-G project

The Condor-G [21] project is a sub project of the Condor. It combines the technologies from the Globus which is the middleware to interconnect multiple domains and from the Condor which concerns of the job submission, job allocation and error recovery. See the following chart.

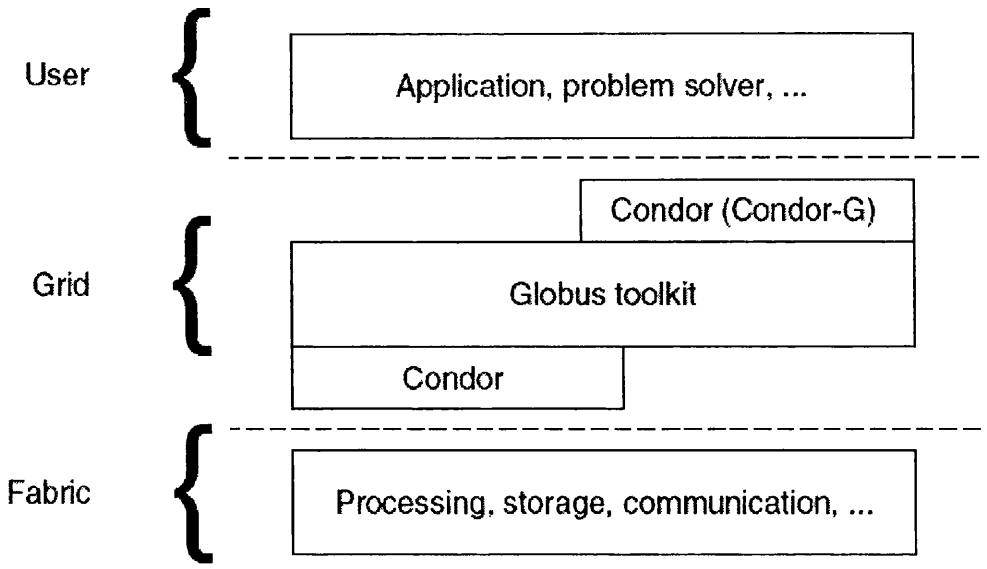

Figure 13

Source: D.Thain, T.Tannenbaum, and M.Livny(2003). "Condor and the Grid, Grid Computing – Making the Global Infrastructure a Reality". John Wiley & Sons, Ltd

The Condor-G locates above the Globus Toolkit (GT) to utilize the GRAM API (mentioned in section 2.3.2) provided by the GT. Since the GT misses some useful features that have been discussed in the Chapter 4, the Condor-G solves these problems by using the technique called "gliding in" which means that the Condor pool's current state is collected by the Condor's agent on the remote site and reported it to the Condor-G which decides where the job should be dispatched. The job is submitted to the Condor pool and the result of the job is transferred back to the user through the GT.

The Condor-G project, like the SCG project, also partially solves the problems mentioned in the previous chapter. It also only allows the LRMS to be the Condor. This reduces the flexibility and adds some limitation on the user's choices.

5.3 The Job Submission Description Language (JSDL) Project

As an ongoing project of the GGF, the JSDL [45] is working on a standard specification to provide the Grid user a unified job description language which makes it simpler to interoperate between the user and LRMS's. The JSDL also puts many efforts to alleviate the difficulty when different job management systems interact with each other.

The JSDL includes three components:

A specification which describes the attributes required when the job is submitted to the Grid system; The XML schema to implement the specification; a set of translation tables to map JSDL attributes and attributes of the LRMS's.

The JSDL project is also motivated by the first scenario addressed in the Chapter 4.1. But it focuses on providing a standard specification which is suggested to be adopted by the Grid applications. How to provide a solution that could reduce the difficulty of submitting the job to LRMS's depends on how to implement the specification by the application developers. The GSV is to provide a whole lifecycle control on the Grid job to solve the mentioned difficulties above rather than a set of specifications.

5.4 The BRIDGES project

The Bridges project [46] is the project which delivers a Grid environment for the biomedical bioinformatics research. The application is developed within the GT3 Grid Service Framework to accommodate the need of concurrent computing from the compute intensive bioinformatics applications.

In this project, the system users also need to execute the job in several existing LRMS's, such as OpenPBS, Condor, etc. So, the researchers of the project developed a GT3 based grid service to implement the BLAST which is "one of the best known sequence comparison programs available in bioinformatics" [47]. One BLAST job is separated to several sub-jobs by the project own scheduler in order to utilize the geographically distributed computational resources. The wrappers of the Condor and the OpenPBS on the remote sites are in charge of the execution and results collection of the BLAST jobs.

The Bridges project is another success project which devotes in the field of the Grid scheduling. Nevertheless, because the project specializes in the bioinformatics area, the scheduler only is limited in processing certain kinds of job type. Of this aspect, the GSV provides a more general solution for the problems mentioned in the previous chapter.

From the discussion above, although there are many attempts to solve the problems that the Grid user encounters in the daily works. The SCG project, the Condor-G project and the Bridges projects all focus on providing the features of

dynamic site's information collection and load balancing of the remote sites. But the first two systems demand the LRMS to be their individual one and the Bridges now can only process specialized Grid applications. The current situation in the Grid community is that many types of the LRMS exist on the resource side and scientists from different fields need to run their jobs on these LRMS's to utilize any LRMS that they can access despite what kind of the LRMS it is. So the GSV designed and implemented a solution to solve all these problems.

Chapter 6 - Methodology

This chapter details the methodologies used in the GSV to address the problems mentioned in the previous chapters. In the GSV, Extended RSL is the single job description language used to hide the differences among the LRMS' job description languages; a super-scheduler architecture that has been proposed by the GGF [40] is the framework of the GSV; and execution site information is periodically gathered by a Provider-Aggregator mechanism.

6.1 Job description language – ERSL

6.1.1 The extended version of RSL – ERSL

Although OGSA has defined a description language (RSL), the attributes in RSL cannot satisfy all the requirements of this project. Therefore, the Grid Scheduling Veneer extends RSL to add the needed, additional attributes.

The ERSL specification is as follows:

RSL String
::= Relation
::= '+' ('(' Relation ')')*
::= '&' ('(' Relation ')')*
Relation
::= 'Variables' '=' (Binding ∞' ')+
::= Attribute ''Op''(Value ∞ '')+
Binding
::= '(' String-Literal ''Simple-Value ')'
Attribute
::= String-Literal

Op

::= '=' | '!=' | '>' | '>=' | '<' | '<='

Value

::= '(' Value ∞‘ ’')'

::= Simple-Value

Simple-Value

::= String-Literal

::= Simple-Value '#' Simple-Value

::= Implicit-Concat

::= Variable-Reference

::= Numeric-String

::= Date-String

::= Boolean-String

Variable Reference

::= '$(' String-Literal ')'

::= '$(' String-Literal ’’ Simple-Value
')'

Implicit-Concat

::= (Unquoted-Literal)? (Implicit-Concat-Core)+

Implicit-Concat-Core

::= Variable-Reference

::= (Variable-Reference)(Unquoted-Literal)

String-Literal

::= Quoted-Literal

::= Unquoted-Literal

Numeric-String

::= Digital+ (’.’Digital+)?

Date-String

::= Digital Digital Digital Digital ‘-‘ Digital Digital
‘-‘ Digital Digital

Boolean-String

::= ‘True’ | ‘False’

Digital

::= ‘0’-‘9’

Quoted-Literal

::= ''' (([^'])|(' ' '))* ' ' '

::= ' " ' (([^"])|(' "" '))* ' " '

::= '^' c(([^c])|(cc))* c

Unquoted-Literal

::= ([^ \t\v\n+&|()=<>!"'^#$])+

Comment

::= '(*' (([^*])|('*'[^)]))* '*)'

Common RSL attributes

arguments

count	stdin
directory	stdout
executable	stderr
environment	queue
jobType	project
maxTime	dryRun
maxWallTime	maxMemory
maxCpuTime	minMemory
gramMyjob	hostCount

Added attributes for the Grid Scheduling Veneer

In order to achieve the GSV system goals, the following attributes were added to the ERSL. These attributes can be classified into three types: System requirements, Job control and Miscellaneous.

Type	Attribute Name	Description
System Requirements	operatingSystem	The OS of the destination system
	archInfo	The architecture of the destination system
	minKflops	The minimum Kflops of the system required for the job
	minMips	The minimum Mips of the system required for the job
	diskSize	Free disk space required
Job Control	systemType	Which LRMS the job prefers
	priority	The job priority in the Grid Scheduling Veneer
Miscellaneous	ssJobID	The job ID generated by the Grid Scheduling Veneer
	sendMail	The notification email address
	localJobDescription	A string transferred directly to the LRMS

The following table describes the format of the added attributes:

operatingSystem ::= String-Literal
archInfo ::= 'intel'|'alpha'|'sun'|'sgi'|'hp'
minKflops ::= Numeric-String
minMips ::= Numeric-String
diskSize ::= Numeric-String
ssJobID ::= String-Literal
systemType ::= String-Literal
priority ::= Numeric-String
sendMail ::= String-Literal
localJobDescription ::= (String-Literal ∞ '\n')+

The System Requirements attributes are used to describe hard constraints on the execution environment required to run the job. The Grid user specifies these requirements in the ERSL string when submitting the job. The GSV matches the execution site's current state with the job's requirements; it then dispatches the job to the selected site or returns a "no qualified site error" to the user.

The Job Control attributes are defined to manipulate the GSV's default scheduling and dispatching rules. The default scheduling algorithm of the GSV is FIFO. The job is inserted into the scheduling queue according to its assigned priority, overriding the default FIFO behaviour. Priority is a numeric value between 0 and 100; larger values represent higher priorities.

The GSV dispatching mechanism load balances the jobs across its execution sites. The System Type attribute affects the choice of site to which a job is dispatched. If this attribute is specified in the ERSL string, the GSV dispatches the job to the user-preferred system in spite of the job load at that execution site.

The ssJobID attribute is the ID of the job assigned by the GSV. Users can control and query jobs by Job ID. The sendMail attribute is used to notify the

user when a job has finished. A message is sent to the mail address specified in the sendMail attribute. The localJobDescription attribute can convey an uninterpreted string to the LRMS. The reason for this attribute is that the GSV only partially achieves the system goal of supporting the superset of the attributes of all three LRMS's. Some attributes understood by the three LRMS's are not included in the ERSL attribute set. Therefore, if the submitter needs to use these attributes, they must be specified in the localJobDescription attribute; the GSV sends this string directly to the LRMS without interpretation.

6.1.2 Job Description language transformation

In the Grid Scheduling Veneer, XML is used as the intermediate language. The Grid Scheduling Veneer needs to transform ERSL to XML, and then to the chosen LRMS-specific language.

The reasons for choosing this model are:

1. There are many existing tools that can be used to parse and interpret XML; for example, Xerces [18] is used to validate the grammar of an XML string. GT3 also provides an API that enables XML construction.
2. The upcoming version of the Globus Toolkit supports XML as the only job description language. Since the Globus Toolkit is the backbone of the GSV project, the GSV needs to anticipate adaptation to the newer version.

The following sections describe the processes of language transformation in the GSV.

1. ERSL to XML

Because the ERSL specification is written in an LALR context-free grammar, it is possible to use a general-purpose parser generator to convert it into a program to parse the grammar, such as Yacc or Bison. Fortunately, Globus provides the APIs to do this, as long as ERSL complies with the RSL specification 1.0. See the following diagram which shows the process of transforming ERSL to XML.

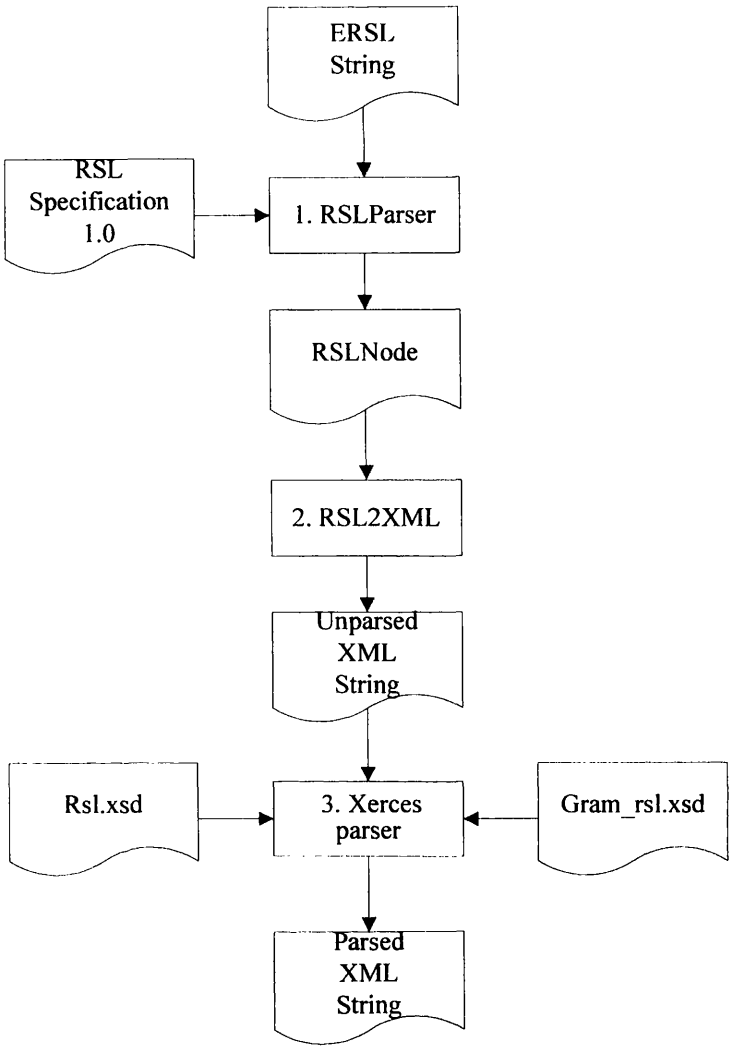

Figure 14

1) **RSLParser:** It uses the API class RSLParser to transform ERSL into an RSLNode, which saves the attribute-value pair (called a relation in RSL) into a Hash Map. In this process, the parser constructs a parse tree from the RSL specification, without validating the RSL string.

2) **RSL2XML:** Obtains the attributes and values from RSLNode objects, constructs XML elements, and adds the predefined XML prefix and suffix to form an unparsed XML string.

3) **Xerces parser:** Xerces[1] parses the XML string generated in the second step and checks its compliance with the schema defined in two files – Gram_rsl.xsd and Rsl.xsd. Finally, a validated XML string is formed.

Rsl.xsd provides the basic type definitions and Gram_rsl.xsd provides the attribute definitions of ERSL.

2. XML to LRMS-specific language

When the job is dispatched to a particular execution site, the XML must be translated to the appropriate LRMS-specific language in order to be submitted to the local system. See the figure below.

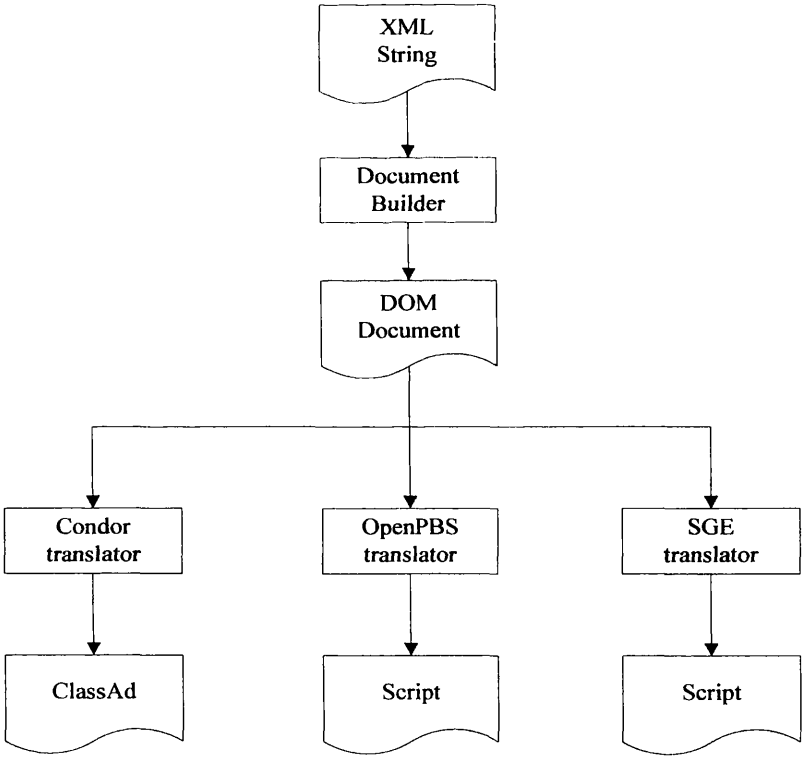

Figure 15

[1] Xerces provides XML parsing and generation and implements the W3C XML and DOM (Level 1 and 2) standards, as well as the de facto SAX (version 2) standard.

1) **Document Builder:** After the Grid Scheduling Veneer has a valid XML string, the string is transformed into an XML Document object, which enables convenient access to the attributes when needed in subsequent stages.

2) **LRMS-specific translator:** Before the job is submitted to the LRMS, the job manager obtains the attributes of the job from the corresponding DOM document and converts them into a Hash table in Perl. Then, the translator, which is written in Perl, obtains the string, analyzes it, and generates the LRMS-specific job description file that is submitted to the particular LRMS.

6.2 Scheduling

Because the Globus Toolkit has no scheduling function, and the GSV dispatches jobs to the multiple execution sites, the GSV must also act as a scheduler to queue jobs submitted by Grid users and dispatch those jobs based on a scheduling algorithm used by the GSV.

6.2.1 Scheduling phases

Grid scheduling is defined as "the process of making scheduling decisions involving resources over multiple administrative domains" [19]. Although the GSV can access multiple domains, it does not own the local resources and has no

control over them. This is the primary differences between an LRMS and the GSV. The GSV must do the following:

1. Discover available resources matching a job's requirements
2. Select the appropriate system(s)
3. Submit the job to the selected system(s).

The author surveyed the Grid literature to find a standard or commonly accepted scheduling proposal that could satisfy the project's requirements. Schopf from the GGF proposed a general architecture for scheduling on the Grid [19]. This proposal not only supports the GSV requirements, but also refines the three primary phases described above into eleven steps. See the figure below.

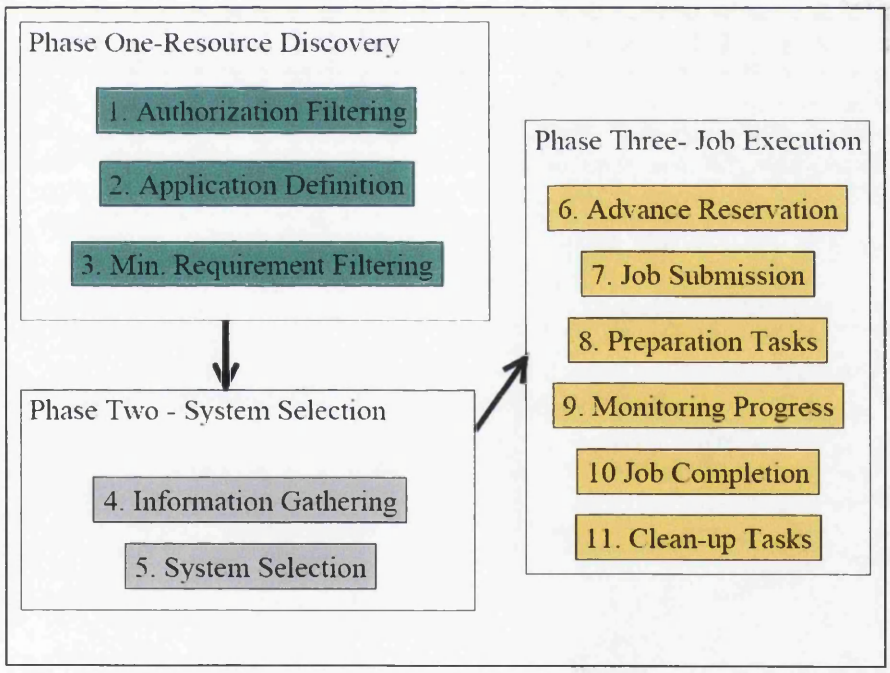

Figure 16

1. Phase One: Resource Discovery

This phase involves determining which resources are available to a given user and selecting a set of resources to be investigated in more detail in Phase Two.

- Authorization Filtering: Determines a resource list that the submitter is authorized to access.
- Application Definition: Retrieves the job attributes from the description string.
- Minimum Requirement Filtering: Works out the minimum requirements of the job.

2. Phase Two: System Selection

Phase Two makes decision where the job should be scheduled by comparing the minimum requirements of the job with the system information gathered.

- Dynamic Information Gathering: Gathers information from the resources on the Grid and stores the information into places easily accessed by the GSV.
- System Selection: With the detailed information gathered in the previous step, the System Selection step decides which resource (or set of resources) to use.

3. Phase Three: Job Execution

This phase includes submitting the job to the selected resource determined in Phase Two, preparing the job's execution environment, monitoring the job execution and clearing up when the job has finished.

- Advanced Reservation: In some cases, the resource needs to be reserved to provide any required QoS.
- Job Submission: Submits the job to the selected resource.
- Preparation Tasks: Builds the job's execution environment including setup, stage in executables and input files and any other preparatory actions required.
- Monitoring Progress: Repetitively queries the status of the job and informs the user.
- Job Completion: When the job has finished, the submitter is notified.
- Cleanup Tasks: Returns the output or error messages and generated files of the job and cleans up the environment that was setup in the Preparation Tasks step.

This general architecture provides a guideline for the Grid Scheduling Veneer, which implements most of the steps described above.

6.2.2 Scheduling and dispatching algorithm

1. Scheduling algorithm

   The scheduling algorithm of the Grid Scheduling Veneer is First-In-First-Out (FIFO) based on job priority.

FIFO: The jobs are ranked by the time when the job is inserted in the queue if the priorities are same.

Priority: When submitted to the Grid Scheduling Veneer, the job should be assigned a priority which is between 0 and 100. By default, the job is assigned the minimum priority value (0) if the priority is unspecified. .

2. Dispatching algorithm

Two kinds of queue exist in the Grid Scheduling Veneer – the private queue and general queue.

General Queue: Holds jobs that can be executed on all types of LRMS's. The General Queue has relatively higher priority than the Special Queue. This means that a job in the General Queue is processed prior to any jobs that are inserted into the Special Queue at the same time.

Special Queue: Each type of LRMS in the Grid Scheduling Veneer resource pool is assigned a queue to hold jobs that are explicitly targeted to this type of LRMS by the user.

See the following diagram.

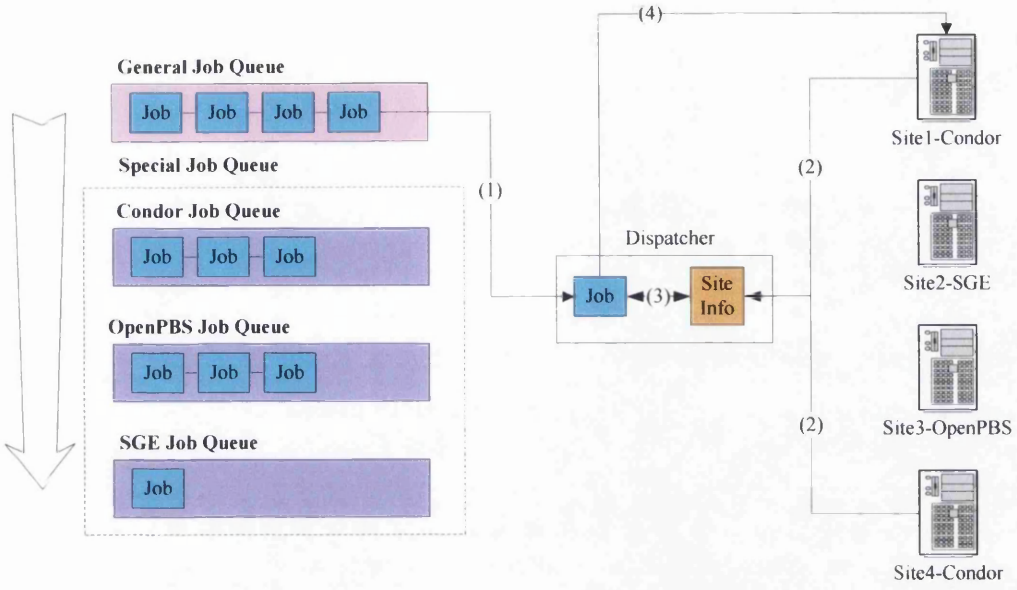

Figure 17

When a scheduling cycle starts, the dispatcher:

(1) Pops the highest priority job from the selected queue, in this case, the General Job Queue.

(2) Gathers the current information from every execution site in the resource pool (Actually, the information is retrieved from the Index Service Database. This is described in the following section).

(3) Chooses a site to which the job should be dispatched if there are multiple sites. The dispatcher considers the following factors ranked in the weight:

a) **LRMS Type**: The user predefined the job's destination LRMS. The job is dispatched to execution sites of the same type.

b) **Hard constraints**: Includes Architecture, Operating System, Memory Size, Disk Size, MIPS and Flops. Only sites meeting all stated hard constraints are eligible.

c) **Queue Size:** The current queue size of the LRMS. If the Queue Size exceeds the value which is the calculated result of the execution site's node number multiplied by a predefined integer number (defined in the GSV's parameter file), this site is labelled as unavailable and is ignored in this scheduling round. The job is dispatched to the site with shortest queue size.

d) **Average System Load:** The dispatcher will route the job to the most lightly-loaded site.

e) If the above factors are same between a number of sites, a site is chosen at random.

(4) Dispatches the job to the selected site.

(5) If all execution sites are all unavailable, ends this scheduling cycle.

(6) If the current queue is not empty,, processes the next element. Otherwise, processes the next queue.

(7) If all queues have been processed, ends this scheduling cycle.

In the GSV, the location where the input data of the Grid job is stored is ignored when dispatching the job although this factor is also a very important criterion since the Grid job always has large datasets' size and the network situation is poor. The job is supposed to be executed on the nearest

execution site in order to minimize the network load. The reason of this absence is because the limitation of the experimental environment where all of the computational nodes are in the same location. Another reason is that the datasets of the job is also transferred with the executables since the data size is comparatively small. But if the GSV is needed to be more practical, the factor of the location of the data should also be considered in the future.

6.3 Execution site information gathering

In the job dispatching process, the dispatcher needs execution site information to make the decision concerning where a particular job should be dispatched. The gathering of execution site information is very critical for job dispatching. The Grid Scheduling Veneer implements the Provider-Aggregator model [43], which is commonly used in information processing systems, to perform this function. See Figure 18.

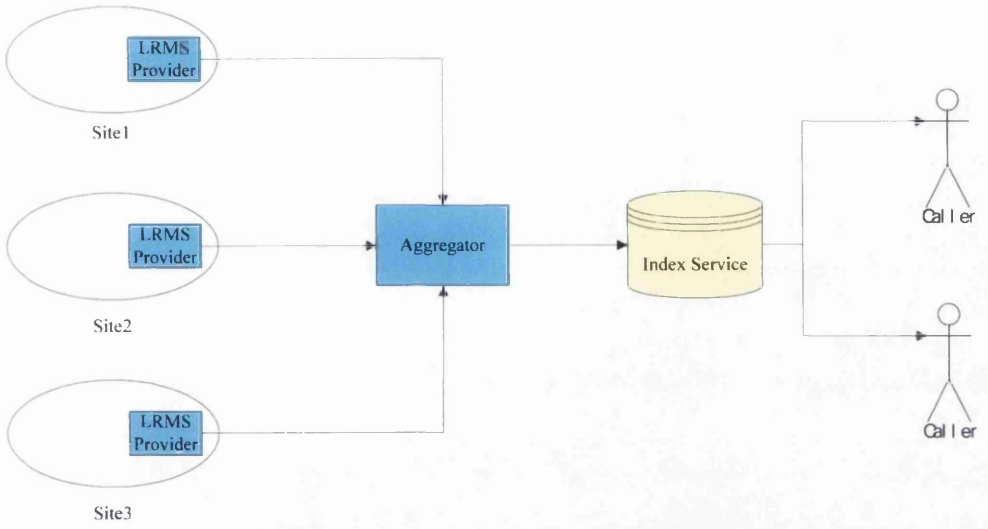

Figure 18

Provider: A process running on the central node of each LRMS gathers current system information, which is encoded in XML. The information gathered includes system Architecture, OS, Memory and Disk size, MIPS, Flops, Queue size and system Load.

Aggregator: A service running in the container along with the Grid Scheduling Veneer collects the information gathered by the Provider processes and stores the information in the Index Service.

Index Service: The Index Service is an information service that uses an extensible framework for managing static and dynamic data for Grids built using the Globus Toolkit 3.2. A caller can access the data in the Index Service through the API it provides.

Chapter 7 - System Design and Implementation

7.1 System Design

The GSV has been designed to hide the complexity of Grid resource scheduling; it provides this value-added function over the Globus Toolkit. The most natural structure for the GSV was as a wrapper (or veneer) of the Globus Toolkit and the three, chosen LRMS's. The following diagram shows the relationship among these three components.

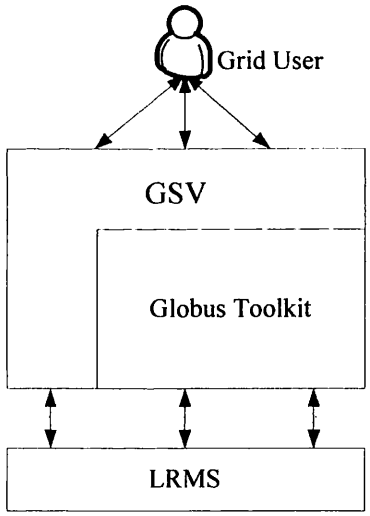

Figure 19

- The GSV interacts with a Grid user to enable job submission, job status determination, and job control. Submitted jobs are dispatched to an LRMS through the Globus Toolkit.

- The Globus Toolkit is the backbone of the GSV. It provides services upon which the GSV depends, such as GRAM and GridFTP, that have been introduced in Chapter 2.
- The LRMS is responsible for execution of a submitted job.

The following figure gives more detail about the system structure of the Grid Scheduling Veneer.

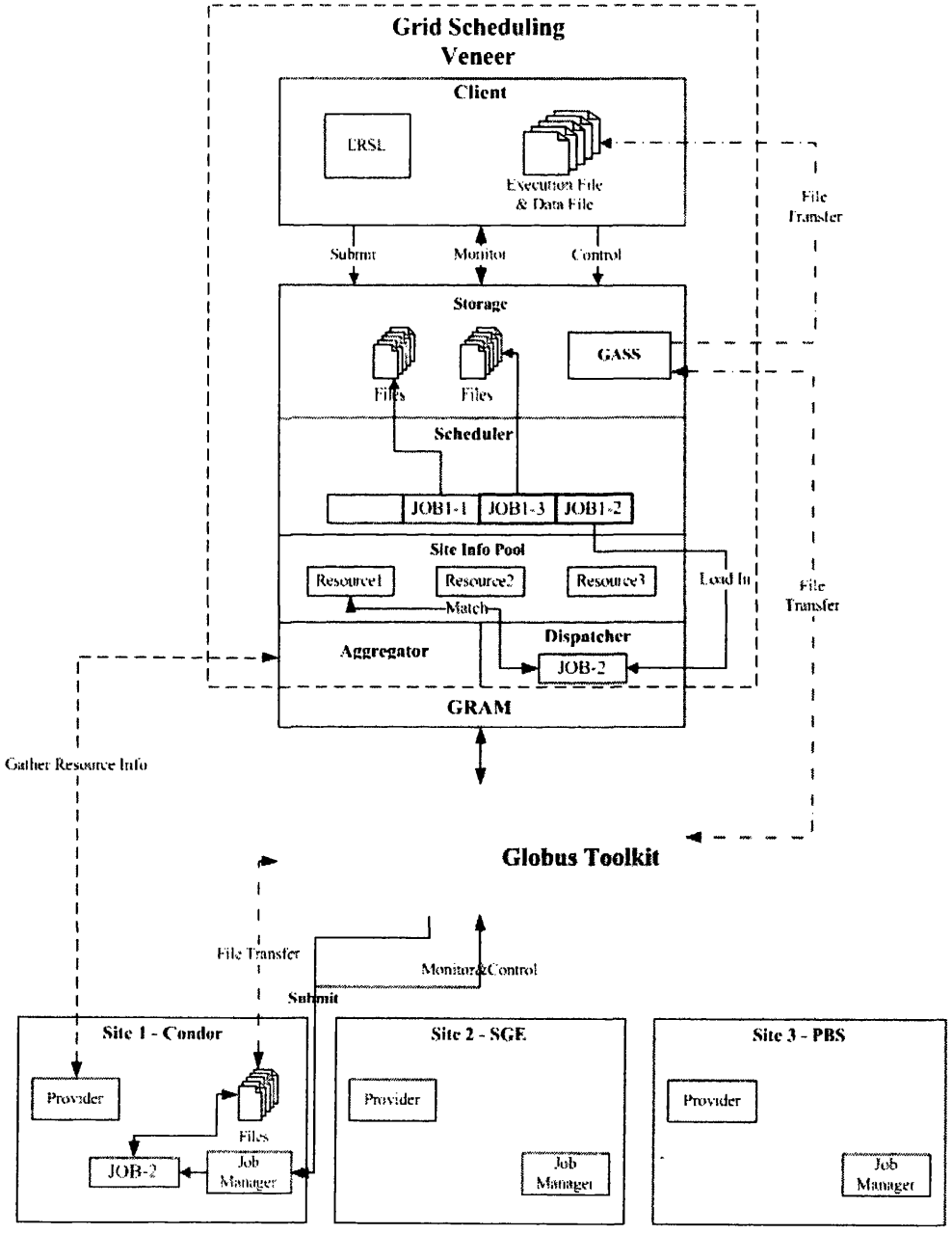

Figure 20

7.1.1 The Grid Scheduling Veneer

The Grid Scheduling Veneer is divided into two parts: the Server Side and the Client Side. The Client Side is installed on any machine which is used to submit, control and monitor jobs.

A) Server side

To meet the system goals, the Server Side should have the following functions:

- Dynamically collects up-to-date information from the execution sites to which it dispatches jobs.
- Provides a scheduling mechanism to queue the submitted jobs.
- Makes dispatching decisions based upon the collected, execution-site information.
- Stages any input files, executables and data files to the chosen execution site, and stages out any results and generated files.

The Server side includes sub-components to achieve these functions.

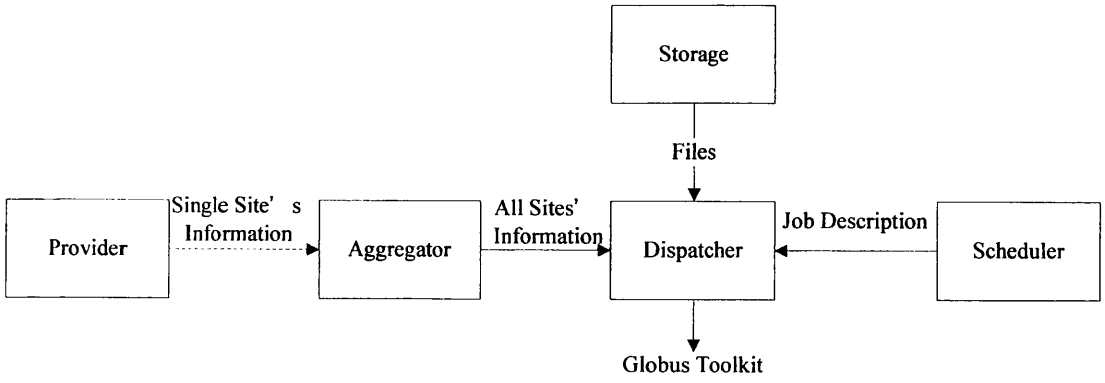

Figure 21

1) Storage:

The Storage component is responsible for input file staging, output file staging, input/output redirection or anything else related to storage. The Grid Scheduling Veneer uses the Global Access to Secondary Storage (GASS) service to provide these functions for a job.

The Grid Scheduling Veneer's client establishes a GASS server when it starts (or is started by the user). When a job is submitted, the user informs the Grid Scheduling Veneer of the job's file locations, and GASS then transfers the job's files including executables, input files and data files to the GASS cache on the execution site. The output or error stream and generated files are delivered back after the job has finished.

2) Scheduler:

The Scheduler component queues submitted jobs, sorting them according to submission time and job priority. The details are in section 6.2.2.

3) Provider:

The Provider component invokes system calls and client tools offered by the LRMS to obtain system information that will be transformed into the required XML format.

4) Aggregator:

The Aggregator iteratively gathers the information from Providers and stores them into the Index Service. Additional discussion about the Provider-Aggregator structure is in section 6.3.

5) Dispatcher:

The Dispatcher component is the keystone of the Grid Scheduling Veneer. It fetches a top priority job from the Scheduler component, analyzes the job attributes, and decides to which site the job should be dispatched by using the aggregated, execution-site information.

The Dispatcher component uses the GRAM API of Globus, discussed in section 2.3.2, to dispatch a job to the execution site.

B) Client side

In order to achieve the goals of scalability and maintainability, a thin-client model was chosen for the design of the Client side. The main function is to provide an interface to enable a user to submit, monitor and control a job.

- Job submission: A job submission client tool is provided by the Grid Scheduling Veneer. Three parameters – the ERSL string or a file containing the ERSL string (required parameter), the GSV Service factory address (required parameter) and GASS Server address (not required parameter) – are supplied when using the submission tool. Each job submitted by the user is assigned a unique job ID that is used to monitor and control the submitted job.
- Job Monitoring: Users can monitor a job's execution status through the client tool's monitor functionality using the job ID parameter. If the job has been dispatched to an execution site, the Grid Scheduling Veneer will try to retrieve the job status through the Globus Toolkit, otherwise it returns the internal status of the job in the Grid Scheduling Veneer's queues.

- Job Control: A user can terminate, suspend or resume a submitted job; if the job has been dispatched, it can only be terminated.

7.1.2 Globus Toolkit

The Globus toolkit has been discussed in section 2.2.1.

7.1.3 Local Resource Management System

The Local Resource Management Systems have been discussed in Chapter 3.

7.2 Implementation

The Grid Scheduling Veneer implements all components as Grid Services. The three Grid services provided by the Grid Scheduling Veneer are the Job Life Control Service Factory Service (JLCSFS) and the Queue Management Service (QMS) on the site where the GSV runs, and the Site Information Provider Service (SIPS) on the execution site.

This structure was chosen for the following reasons:

1. There could be many users accessing the GSV simultaneously. Each user possesses an instance of JLCSFS to hold the state of that user's submitted jobs, thus providing natural isolation between jobs of different usesrs. The instance is destroyed when the user's job has finished.

2. Since all jobs from different users are gathered and dispatched through one dispatcher, only one instance of the QMS, which is in charge of matchmaking and dispatching, is needed.

3. The SIPS runs on the execution site in order to access that site's current information directly.

The following diagram shows the internal structure of the GSV.

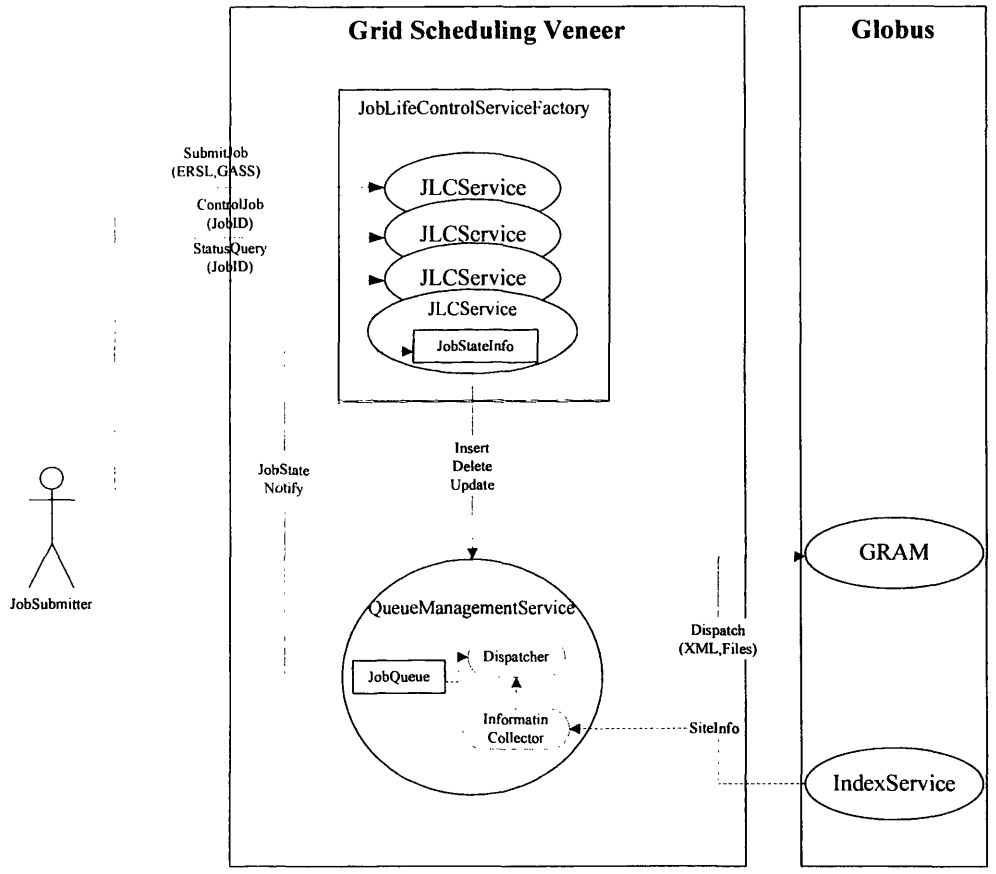

Figure 22

7.2.1 Job Life Control Service Factory

The Job Life Control Service Factory instantiates a Job Life Control Service when a user invokes the Job Submit method of the client tool, and that instance

exists until the job has finished. In other words, there is a one to one relationship between a job and its Job Life Control Service.

1) Interface:

- SubmitJob: To submit the job to the Grid Scheduling Veneer.
- QueryJob: To obtain the current status of the job.
- ControlJob: To suspend, resume or terminate a submitted job.

2) Key process flowchart:

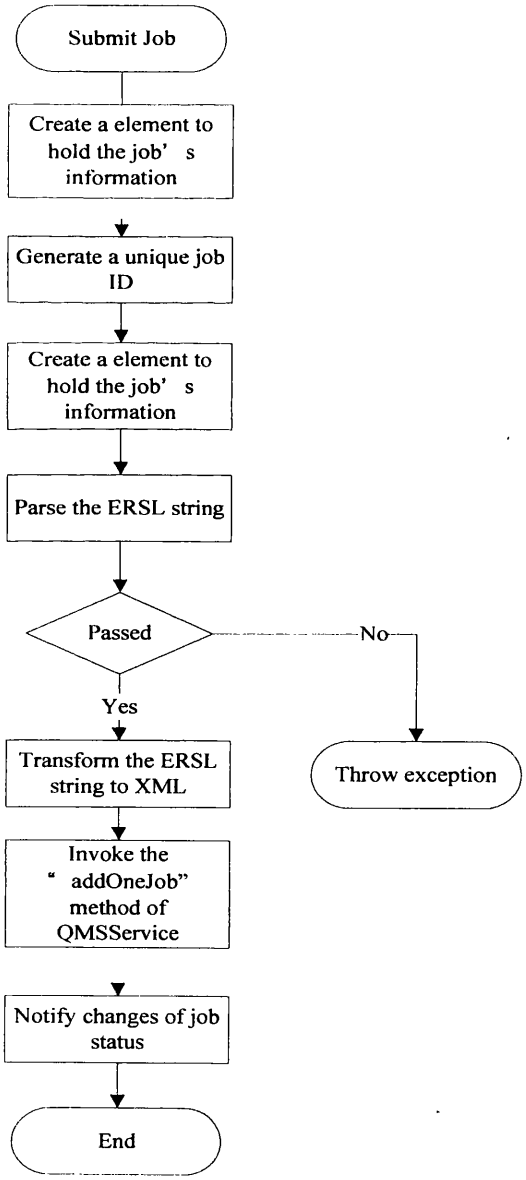

Figure 23

7.2.2 Queue Management Service

The Queue Management Service is a persistent service which is initialized by the Grid Service container. Therefore, only one service instance exists in the Grid Scheduling Veneer to undertake the job of queue management and dispatching.

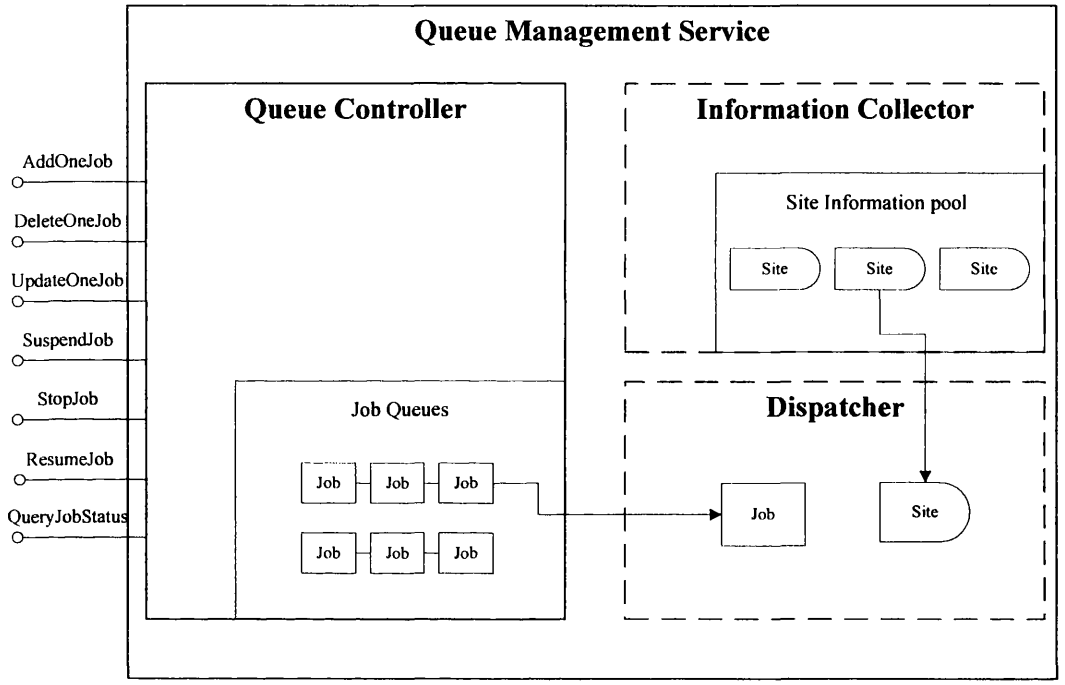

Figure 24

Three components comprise the Queue Management Service - i.e., Queue Controller, Information Collector and Dispatcher.

- Queue Controller

  The Queue Controller is in charge of initialization and management of the job queues. It runs in the main thread of the service and interacts with the other two components.

- Information Collector

  Gathers the execution sites' information stored in the Index Service and buffers them in the information pool. It runs as a timed task which is executed periodically in a child thread.

- Dispatcher

  Makes a match between a job and an execution site and dispatches the job to the selected site.

  Initially, the dispatcher was implemented as an event driven task that starts a scheduling cycle after receiving a notification from the Queue Controller. But the notification mechanism in GT3 is somewhat unstable. Notifications often experience lengthy delays, and sometimes they are lost. Therefore, the Dispatcher was changed to a timed task in a child thread; it should be changed back if the Globus Toolkit ever provides a reliable notification mechanism in a future version.

1) Interface

   - AddOneJob: To insert a job into the queue.
   - DeleteOneJob: To remove a job from the queue
   - UpdateOneJob: To update the attributes of a job in the queue.
   - SuspendJob: To suspend a job in the queue and move it to the suspended job queue.
   - ResumeJob: To resume a suspended job and move it back to the original job queue.

- StopJob: To stop a job in the queue or already submitted to the execution site.
- QueryJobStatus: To query a job's current status.

2) Key process flowchart:

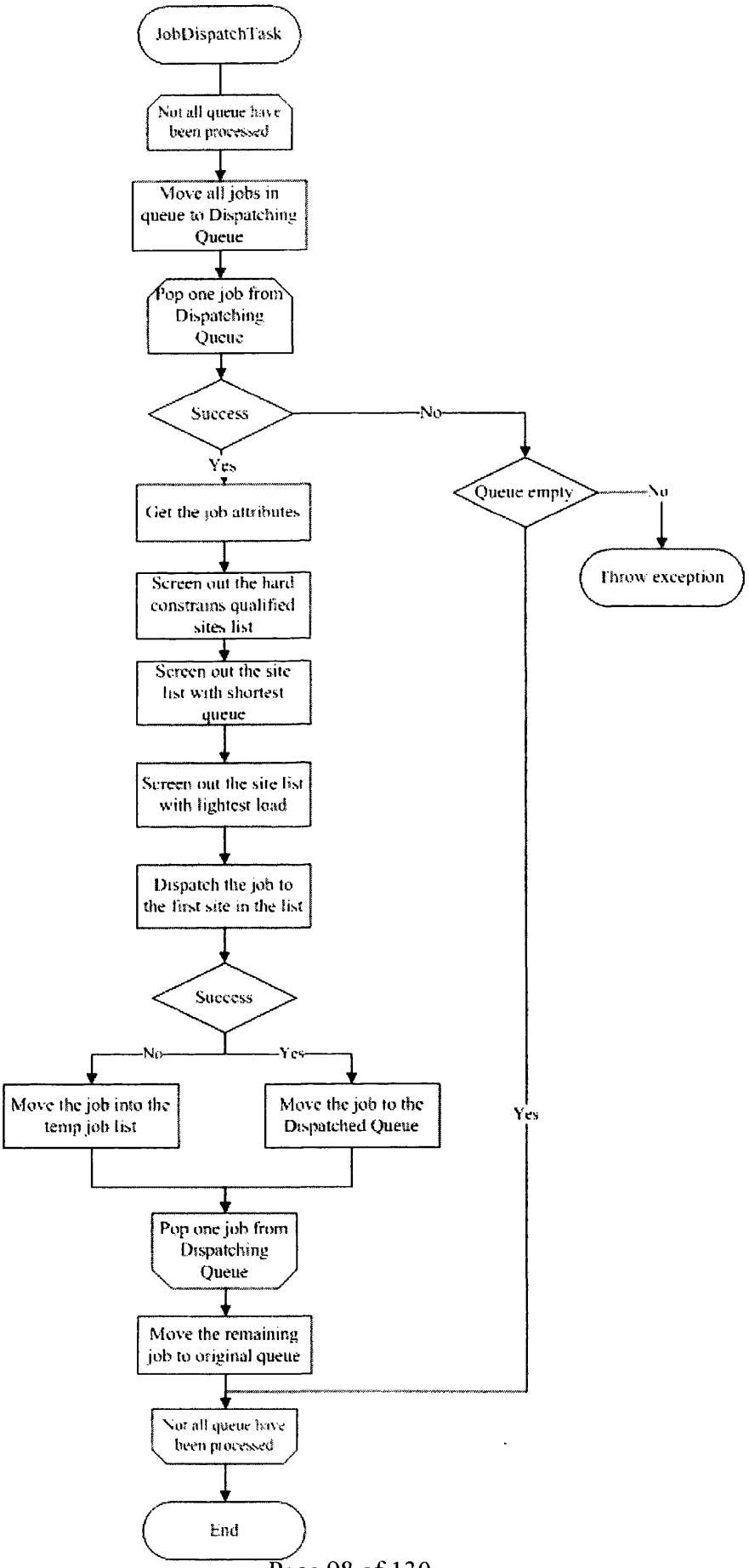
JobDispatchTask
Not all queue have been processed
Move all jobs in queue to Dispatching Queue
Pop one job from Dispatching Queue
Success
No
Yes
Queue empty
No
Throw exception
Get the job attributes
Screen out the hard constrains qualified sites list
Screen out the site list with shortest queue
Screen out the site list with lightest load
Dispatch the job to the first site in the list
Success
No
Yes
Move the job into the temp job list
Move the job to the Dispatched Queue
Yes
Pop one job from Dispatching Queue
Move the remaining job to original queue
Not all queue have been processed
End

Figure 25

7.2.3 Site Information Provider

A key issue for successful job scheduling is that the system is able to obtain up-to-date system information from the LRMS's. To that end, a Site Information Provider runs on each execution site to collect the required system information. Currently, there are three providers – Condor Provider, OpenPBS Provider and SGE Provider – to gather information from the three supported LRMS's. The common interface to the SIPS enables the heterogeneity of the supported LRMS's to be hidden.

Two kinds of system information are gathered by the Providers.

- Static Information: Includes the LRMS's Architecture, Operating System, Memory Size, Flops and MIPS
- Dynamic Information: Includes the LRMS's current Queue Size, Average System Load, Free Disk Size and Number of Execution Nodes.

Static information is obtained as the service starts up and will not change during the lifetime of the service. Dynamic information is refreshed regularly. The site information is pushed to the GSV periodically.

7.3 Experimental Design

Upon completion of the GSV implementation, it was subjected to a series of experiments to evaluate whether the system goals had been met.

7.3.1 Experimental goals

The experiment is designed to ascertain if the Grid Scheduling Veneer fulfils the system requirements and to study the behaviour of the Grid Scheduling Veneer and the three LRMS's when they run different kinds of job streams. The experiment may also suggest improvements to the scheduling algorithm based on the results measured.

7.3.2 Experimental environment

Ideally, the topology of the environment should look like the following diagram. Three clusters, one for each of the supported LRMS's, form the target execution environment. This topology guarantees that there is no interference between the separate clusters..

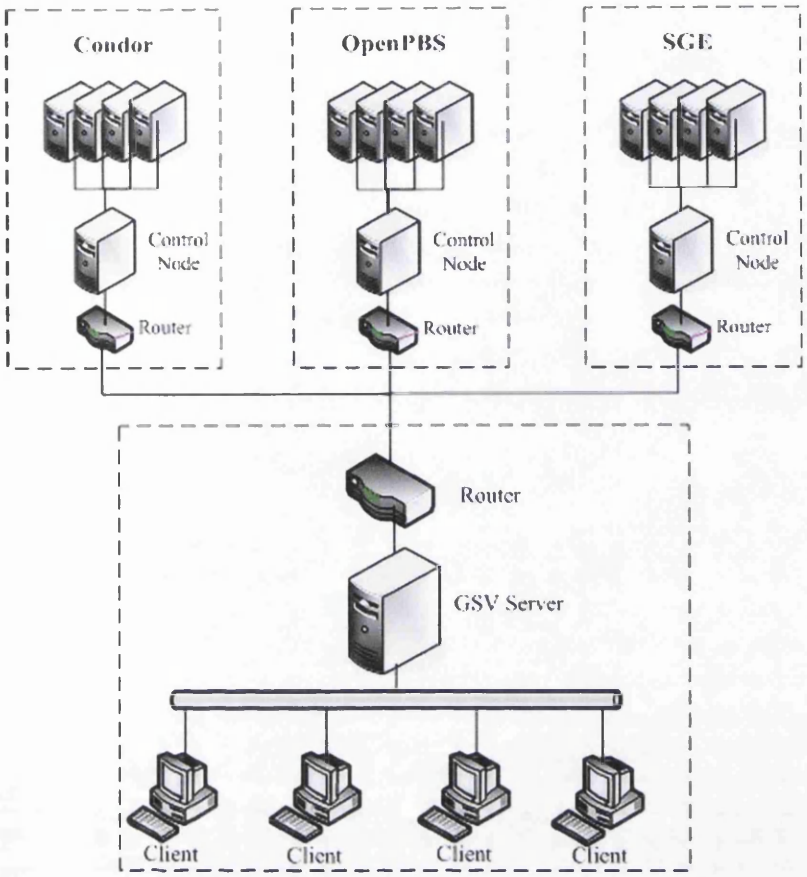

Figure 26

7.3.2.1 Topology

Since the project primarily focuses on functionality rather than performance, four machines were involved in the experiments (See the figure below). One of them was assigned as a Control Node and the others were Execution Nodes. Some services are also needed in the environment, such as DNS, NIS and NFS; this forced some machines to take on multiple roles.

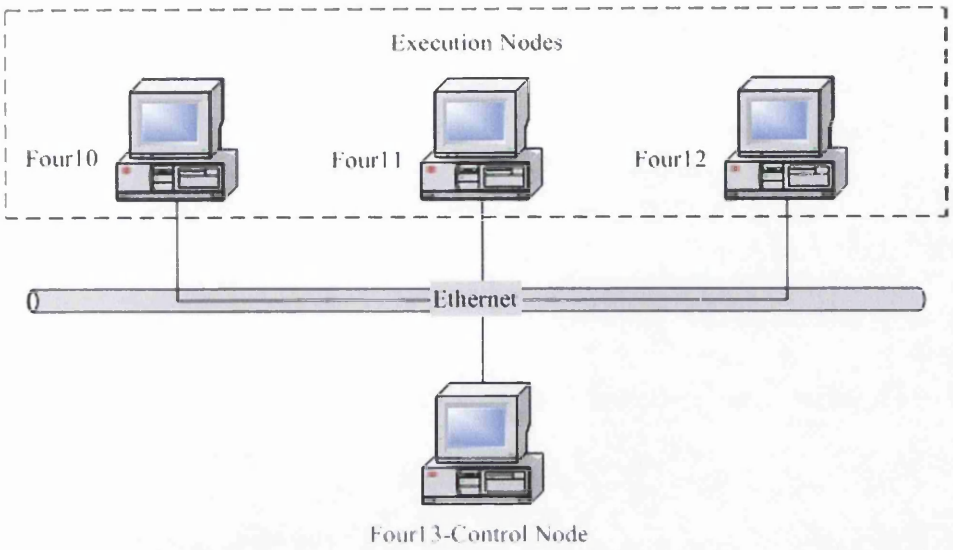

Figure 27

7.3.2.2 Hardware information

Hostname	CPU	Memory size	Disk size
Four10.four	Pentium II 366MHz	256MB	6GB
Four11.four	Pentium II 366 MHz	256MB	6GB
Four12.four	Pentium II 333 MHz	256MB	20GB
Four13.four	Pentium II 333 MHz	128MB	8GB

Unfortunately the machines are not identical, thus introducing some variability in the results of the experiments. But this is also a benefit because the differences among these machines construct a heterogeneous experimental environment which is a basic attribute of the real Grid world.

7.3.2.3 Software information

Hostname	Duty	Software
Four10.four	DNS Server, NIS Server, Execution Node	RH Linux 9.0, BIND 9.2.3, YPServ 2.8, Condor node, OpenPBS node, SGE node
Four11.four	Execution Node	RH Linux 9.0, Condor node, OpenPBS node, SGE node
Four12.four	Execution Node	RH Linux 9.0, Condor node, OpenPBS node, SGE node
Four13.four	File Server, Control Node	RH Linux 9.0, GT3.2, Condor Server, OpenPBS Server, SGE Server

Figure 28

7.3.3 Three stages of the experiment

The experiment is divided into three stages.

Stage 1: Only run one LRMS on the systems and submit the job streams to the LRMS directly (i.e. without using the Grid Scheduling Veneer). The cluster configuration is shown in Figure 29 (with Condor as an example). From this stage, we obtain a baseline for each LRMS to compare with later results.

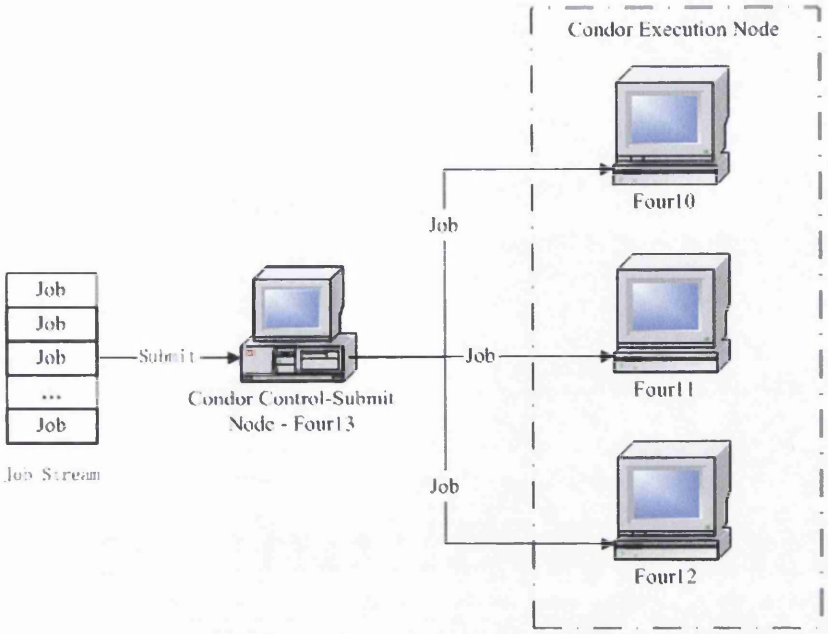

Figure 29

Stage 2: The job streams are submitted through the Grid Scheduling Veneer to only one of the three LRMS's (again, Condor is shown as an example).

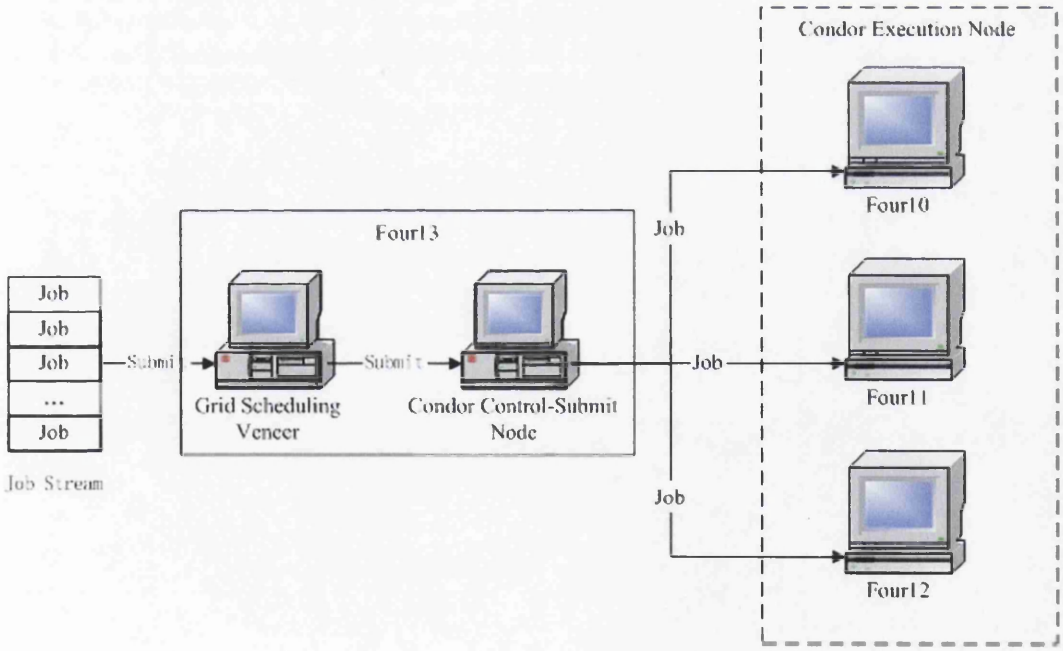

Figure 30

Stage 3: The Grid Scheduling Veneer dispatches jobs to all three LRMS's, each of which controls one node.

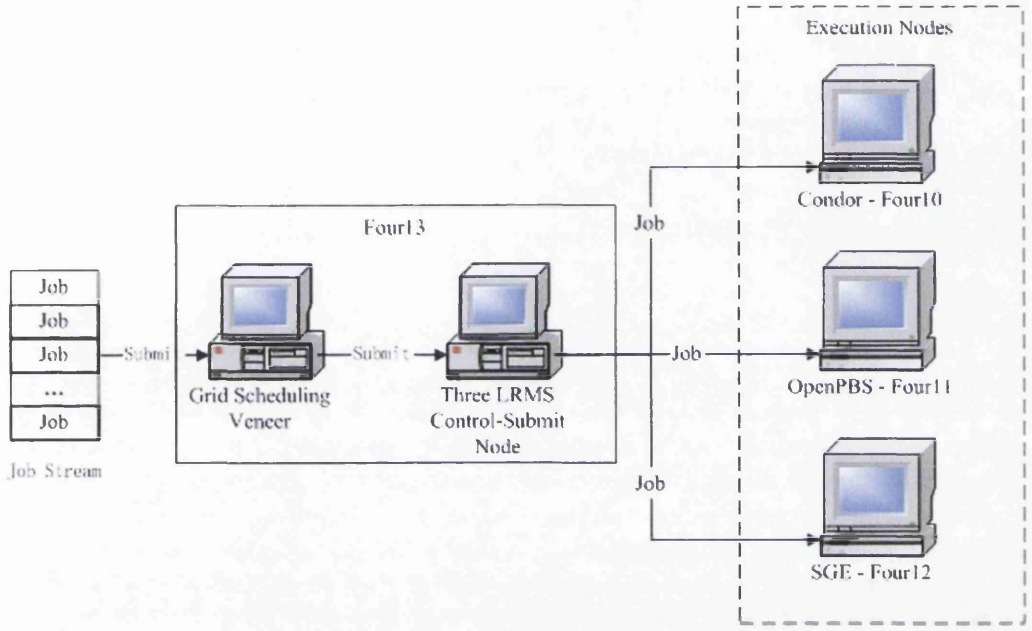

Figure 31

For each run, each job stream is submitted four times, and the average execution time for the stream is calculated.

7.3.4 The experiment to study the overhead of the GSV

Another experiment is designed to determine the overhead of the GSV. Because the three execution nodes were not identical, only one execution node of the Condor cluster was involved in this experiment. See the following diagram.

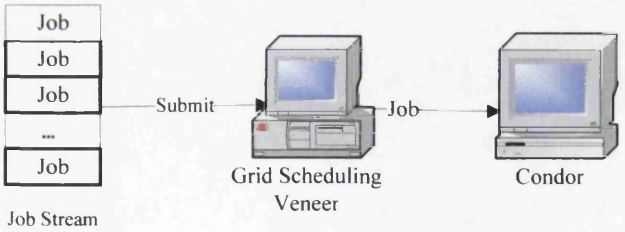

Figure 32

For this set of experiments, seven job streams were used: one with a single job, the second with two instances of the same job, ..., the seventh with seven instances of the same job.

First, we submitted each job stream directly to the Condor system. After obtaining the execution time of each stream, we submitted each job stream through the GSV, and then compared the two results to obtain the overhead of the GSV and the trend of the overhead as a function of the number of jobs in the stream.

In this experiment, because there is one execution site, Condor, which has one execution node, only one job can be executed at a time. This avoids variability of the results which would be induced by use of non-identical experimental computers.

We chose a single, sample job to form job streams of various lengths. If the execution time of a single job in the target LRMS is a constant value (this has been validated in the experiment, see figure 56, 57), the total execution time of the experimental job when the GSV is involved can be divided into two parts – the processing time of the GSV and the execution time of the LRMS. Moreover, the processing time of the GSV also includes two parts – the processing time before being inserted into the LRMS queue and the processing time thereafter. All of the jobs in a particular job stream are inserted into the GSV at one time; as a result, the GSV service instance (discussed in the section 7.2) is initialized once; additionally, the sum of a job's waiting time in the queue and dispatching time is

much bigger than the processing time prior to the insertion of the job into the queue (discussed in section 8.3.2). Therefore, the processing time of the GSV mostly depends on each individual job's queuing and dispatching time. So in the following discussion, the each job's processing time of the GSV is considered as a single variable.

See the following diagrams.

1. The state of the system when the job stream has been submitted to the GSV.

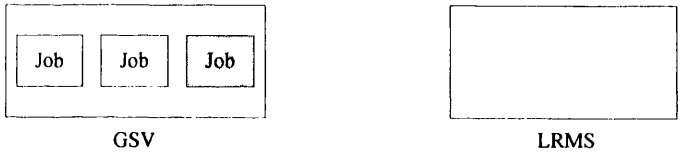

Figure 33

The LRMS is waiting for the GSV processing of the current job (shaded in the figure) to complete. Because the job is processed only after the previous job has been dispatched, the next job is just waiting for the completion of the previous processing job. Say the current job's GSV processing time is Tg. So the LRMS will execute the first submitted job after Tg1.

2. The state when the LRMS has received the first job

Figure 34

The received job is executing in the LRMS. The execution time of that job is T_{LRMS} and the second job's processing time in the GSV is Tg2.

3. If Tg2 is less than or equal to T_{LRMS}:

Figure 35

The queue of the LRMS has accumulated one extra job.

1) If the processing time of the GSV is a constant or always shorter than T_{LRMS}, the LRMS queue will never be empty. The whole remaining execution time only depends on the execution time of the LRMS. So the GSV overhead is a constant i.e. Tg.

2) If the GSV overhead per job is not a constant, the situation is more complex. The overhead is a function which depends on two variables, i.e. the current job's processing time in the GSV and the whole jobs' executing time in the LRMS' queue. If the processing time is larger than the executing time, the overall overhead increases with the difference of the two values, otherwise the overhead stays flat. Thus the overhead of the GSV is not a linear function. Whether the processing time is constant or not could be discovered in the experiment.

4. If Tg2 is bigger than T_{LRMS}:

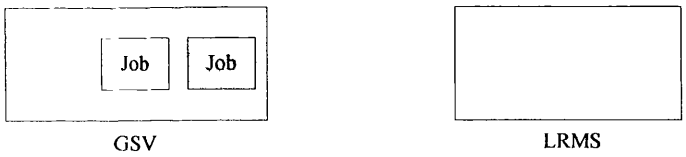

Figure 36

The LRMS queue will be empty, waiting for the arrival of the next job.

1) If the processing time of the GSV is a constant or always larger than T_{LRMS}, the LRMS will always wait for the next job. The whole remaining execution time depends on the difference of the processing time of the GSV and the T_{LRMS}. As a result, the overhead is a linear function related to the job number in the stream. See the expression in Figure 41.

2) The situation when the processing time of the GSV is not a constant is same to the one in the discussion of the figure 35. It is not a linear function.

The following paragraphs derive an equation for the overhead introduced by the GSV.

- If the processing time of the GSV is a constant value T_g, the $T_{Total}(n)$ can be represented as:

$$T_{Total}(n) = T_g + T_{LRMS} + \sum_{i=2}^{n} \max(T_{LRMS}, T_g)$$

Figure 37

where:

T_{Total}: The total execution time of the job stream in the GSV

T_g: The processing time of single job which also includes the network communication time before the job is submitted to the LRMS.

T_{LRMS}: The execution time of a single job in the LRMS

n: The current job number

Let Over(n) be the overhead expression for the n^{th} job in the stream, and J be the total number of jobs in the stream.

- If T_g is less than T_{LRMS}, the expression becomes

$$T_{Total}(J) = T_g + J * T_{LRMS} :$$

Figure38

The overhead of the GSV can be calculated as:

$$Over(J) = T_{Total}(J) - J * T_{LRMS} = T_g + J * T_{LRMS} - J * T_{LRMS} = T_g$$

Figure39

Thus, the overhead of the GSV should be a constant value, equal to T_g.

- If T_g is greater than T_{LRMS}, the expression becomes:

$$T_{Total}(J) = T_{LRMS} + J * T_g$$

Figure40

The overhead of the GSV can be calculated as:

$$Over(J) = T_{Total}(J) - J * T_{LRMS} = T_{LRMS} + J * T_g - J * T_{LRMS} = T_{LRMS} + J * (T_g - T_{LRMS}$$

Figure41

Thus, the overhead of the GSV should scale linearly with the number of jobs in the stream.

7.3.5 Job streams

In order to make the experiments somewhat realistic, we collected eleven applications from scientists who use the Grid for their research. Each binary is compiled statically to avoid the need to set up the execution environment on every machine in the cluster. The Type column below indicates whether the application is CPU-intensive or I/O-intensive.

Appl	Description	Type	Executable (Byte)	Input File Size(Byte)	Generated FileSize(B)	Output Size(B)
Gadget	cosmological simulations	CPU	69,170	62,593	9,055,518	1,035,495
Eprun	Monte Carlo event generatorused in particle physics	CPU	20,170,888	10,320	0	1,039,707
Lisa	The Laser Interferometer Space Antenna	I/O				
	X-Setup	I/O	75,918	0	336,855,040	2,530
	Y-Setup	I/O	75,918	0	336,977,920	2,530
	Z-Setup	I/O	75,918	0	336,855,040	2,530
NS2	Network Simulator2					
	Simple	I/O	5,642,302	1,517	289,585,175	26
	Fq	I/O	5,642,302	8,663	204,582,579	328
	Ss	I/O	5,642,302	1,505	151,055,134	61

	Large	CPU	5,642,302	127,202	0	1,509
	Ranvar	CPU	5,642,302	964	0	650,236
Xmd	Molecular dynamics simulator	CPU	291,864	3,786	8,085,936	143,819

Figure 42

Three streams are constructed from these applications.

CPU intensive stream	Large, Ranvar, Gadget, Eprun and Xmd
I/O intensive stream	Simple, Fq, Ss, X-Setup, Y-Setup and Z-Setup
MIX stream	Large, Simple, X-Setup, Y-Setup, Gadget, Eprun and Xmd

Figure 43

Chapter 8 - Results

8.1 The results of three-stage experiment

The following tables record the results of the three-stage experiment.

8.1.1 Stage 1 – Baseline Measurements

1) Raw data

	Condor			OpenPBS			SGE		
	Start	Stop	Length	Start	Stop	Length	Start	Stop	Length
MIX	17:46:46	20:09:10	2:24:24	19:40:19	22:11:41	2:31:22	12:51:25	15:32:38	2:48:23
	20:19:44	22:42:18	2:22:34	23:04:37	1:32:59	2:28:18	17:10:16	19:51:18	2:41:02
	23:33:16	1:54:41	2:38:35	1:58:33	4:28:47	2:30:14	22:10:17	0:52:12	2:41:55
	10:49:35	13:15:26	2:25:51	9:31:20	12:03:08	2:31:48	1:25:51	4:13:32	2:47:41
CPU	17:14:58	19:19:19	2:04:21	14:49:17	16:57:29	2:08:12	10:58:23	13:04:14	2:03:51
	19:39:30	21:44:07	2:04:37	17:30:12	19:41:09	2:10:57	16:55:14	19:02:50	2:07:36
	22:35:03	0:39:45	2:04:42	21:53:39	00:01:58	2:08:17	15:31:44	17:35:23	2:03:41
	21:41:41	23:46:27	2:04:46	0:15:51	2:25:13	2:09:22	17:50:19	19:55:32	2:05:13
IO	21:43:34	23:46:57	2:03:23	17:04:40	19:18:52	2:14:12	12:24:57	14:56:53	2:31:56
	23:59:29	2:02:53	2:03:24	19:59:28	22:11:32	2:12:04	17:39:16	20:06:57	2:27:41
	2:07:51	4:16:34	2:08:43	22:37:51	00:51:03	2:13:12	21:23:35	23:52:41	2:29:06
	10:55:21	12:58:56	2:03:35	1:25:29	3:35:46	2:10:17	0:45:02	3:14:51	2:29:49

Figure 44

2) Summary

Stream	Condor	OPENPBS	SGE	Average
CPU	2:04:33	2:09:12	2:05:03	02:06:16
I/O	2:04:46	2:12:26	2:29:38	02:15:37
MIX	2:27:51	2:30:26	2:44:45	02:34:21

Figure 45

8.1.2 Stage 2 – Single LRMS with GSV

1) Raw data

	Condor			OpenPBS			SGE		
	Start	Stop	Length	Start	Stop	Length	Start	Stop	Length
MIX	15:26:50	18:17:31	2:50:41	20:54:07	23:39:00	2:44:53	0:47:08	3:39:38	2:52:30
	20:08:33	22:59:53	2:51:20	9:10:03	11:56:22	2:46:19	9:12:28	12:03:53	2:51:25
	23:15:55	2:08:58	2:53:03	12:48:07	15:27:54	2:39:47	12:29:03	15:24:42	2:55:39
	8:38:12	11:27:46	2:49:34	12:33:32	15:25:50	2:52:18	15:33:41	18:26:09	2:52:28
CPU	12:44:08	14:57:56	2:13:48	0:04:18	2:20:53	2:16:35	15:32:30	17:50:45	2:18:15
	15:19:53	17:37:21	2:17:28	12:29:10	14:48:22	2:19:12	18:23:51	20:42:25	2:18:34
	18:25:18	20:38:27	2:13:09	15:31:41	17:46:06	2:14:25	21:39:33	23:58:51	2:19:18
	21:05:24	23:17:28	2:12:04	18:19:19	20:34:49	2:15:30	1:19:09	3:39:47	2:20:38
IO	16:00:31	18:13:22	2:12:51	1:13:44	3:49:10	2:35:26	17:59:15	20:31:55	2:32:40
	19:11:47	21:25:29	2:13:41	10:04:32	12:44:39	2:40:07	21:07:51	23:41:09	2:33:18
	0:09:29	2:21:58	2:12:29	12:58:10	15:35:41	2:37:31	0:11:29	2:42:55	2:31:26
	2:35:19	4:45:57	2:10:38	16:09:29	18:45:52	2:36:23	14:20:46	16:53:18	2:32:28

Figure 46

2) Summary:

Stream	Condor	OPENPBS	SGE	Average
CPU	2:14:07	2:16:26	2:19:11	02:16:35
I/O	2:12:25	2:37:22	2:32:28	02:27:25
MIX	2:51:09	2:45:49	2:53:00	02:50:00

Figure 47

8.1.3 Stage 3 – GSV scheduling all three LRMS's

1) Raw data

Stream	Start	Stop	Length	Stream	Start	Stop	Length
MIX	09:23:26	12:19:30	02:56:04	**I/O**	01:11:36	03:32:55	02:21:19
	13:03:49	15:43:12	02:39:23		10:10:42	12:34:09	02:23:27
	15:49:42	19:23:31	03:33:49		14:28:05	14:42:39	02:14:34
	21:25:38	00:47:50	03:22:12		15:03:32	17:24:37	02:21:05
	10:20:52	12:46:09	02:25:17		19:13:51	21:27:31	02:13:40
	13:00:07	16:30:29	03:30:22		21:49:21	00:21:09	02:31:48
	18:32:37	20:57:53	02:25:16		15:03:32	17:24:39	02:21:07
	21:03:13	23:58:47	02:55:34		17:35:51	20:06:28	02:30:37
CPU	14:00:40	16:09:41	02:09:01				
	16:39:17	18:49:38	02:10:21				
	19:29:02	21:38:10	02:09:08				
	22:09:25	00:21:46	02:12:21				

Figure 48

2) Summary

Stream	Execution Time
CPU	02:10:13
I/O	02:22:12
MIX	02:49:07

Figure 49

8.2 The results of overhead experiment

In the experiments, one application – "Large of NS2" – was chosen as the sample job. The reason to choose this application is that it is easy to control the execution time by changing the parameters and input dataset size. Thus, the experiment could show the different behaviours of the GSV when the single job's LRMS execution time is longer than the processing time of the GSV or vice versa.

When $T_{LRMS} > T_g$, the following table results:

	One job	Two jobs	Three jobs	Four jobs	Five jobs	Six jobs	Seven jobs
Execution Time Without the GSV	1:07	2:10	3:15	4:21	5:36	6:30	7:32
	1:07	2:10	3:15	4:23	5:25	6:27	7:30
	1:07	2:09	3:15	4:34	5:22	6:25	7:29
	1:06	2:10	3:14	4:18	5:26	6:25	7:34
Average	1:06	2:09	3:14	4:24	5:27	6:26	7:31
Execution Time With the GSV	1:56	3:01	4:20	5:22	6:18	7:16	8:38
	1:56	2:59	4:08	5:14	6:30	7:25	8:26
	1:55	3:00	4:15	5:18	6:14	7:23	8:15
	1:56	3:03	4:12	5:10	6:20	7:15	8:30
Average	1:55	3:00	4:13	5:16	6:20	7:19	8:27

Figure 50

The following table shows the results when $T_g > T_{LRMS}$.

	One job	Two jobs	Three jobs	Four jobs	Five jobs	Six jobs	Seven jobs
Execution Time Without the GSV	0:56	1:45	2:43	3:40	4:40	5:33	6:29
	0:55	1:50	2:43	3:41	4:33	5:33	6:24
	0:56	1:48	2:45	3:37	4:28	5:38	6:23
	0:56	1:47	2:47	3:39	4:26	5:37	6:32
Average	0:55	1:47	2:44	3:39	4:31	5:35	6:27
Execution Time With the GSV	2:11	3:30	4:38	6:02	7:36	8:30	10:13
	2:17	3:28	4:45	6:06	7:15	8:28	9:53
	2:15	3:27	4:39	5:56	7:28	8:50	9:58
	2:08	3:22	4:33	6:05	7:33	8:46	9:40
Average	2:12	3:26	4:38	6:02	7:28	8:38	9:56

Figure 51

8.3 The results analysis

The experimental results from the three stages show: 1) the LRMS's exhibit behaviour diversity when dealing with different kinds of jobs (CPU intensive and I/O intensive), and 2) performance differences exist between each LRMS and the Grid Scheduling Veneer.

8.3.1 The performance difference of three LRMS's

From the stage one results, we can get an overall image of the performance difference of the three LRMS's. See the following figure.

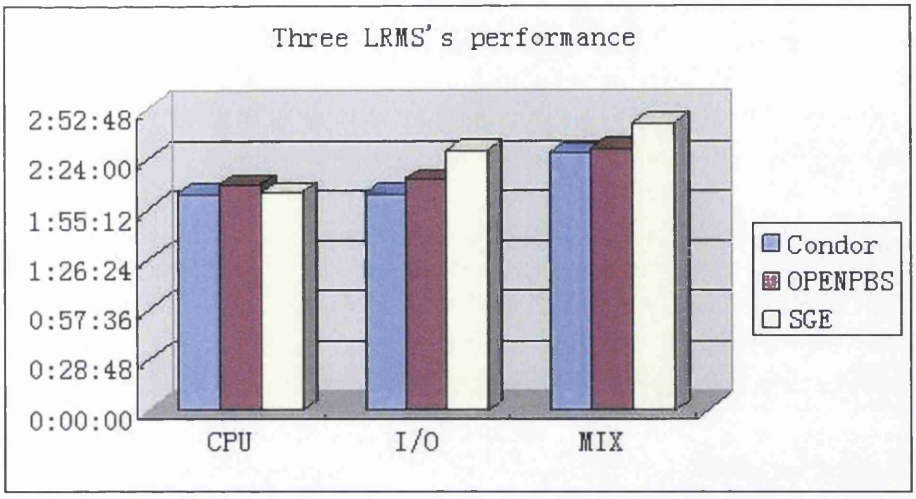

Figure 52

Condor's overall performance is the best among the three systems, especially for the I/O intensive job stream. The results benefit from the file transfer mechanisms of Condor. Condor works very well without a shared file system. It automatically transfers the input files from the submitting machine into a temporary working directory on the execution machine. As a result, all file read/write operations happen locally. In contrast, OpenPBS and SGE depend on a shared file system to transfer the input and output files. This is a bottleneck that slows down the I/O performance.

A serial of tests was taken to examine the I/O performance of the experimental machines. Because the file system server is also an execution node in the experiment, the serial of tests includes one server test and two kinds of client test, i.e. one is the reading and writing speed of the clients without the server reading or writing simultaneously and another one is with the server. The results show the different situations, one is to simulate

Condor I/O processing whose I/O operation does not include the server and the later one is the OpenPBS and the SGE whose include the server. See the following charts which show the I/O performance of the experimental environment.

The first chart shows the performance when the client and server read data files which are stored on the file server. The trend lines show the reading speeds of server, client1 (server reads simultaneously), and client2 (server is not involved) respectively. The second diagram illustrates the write performance in the same situation.

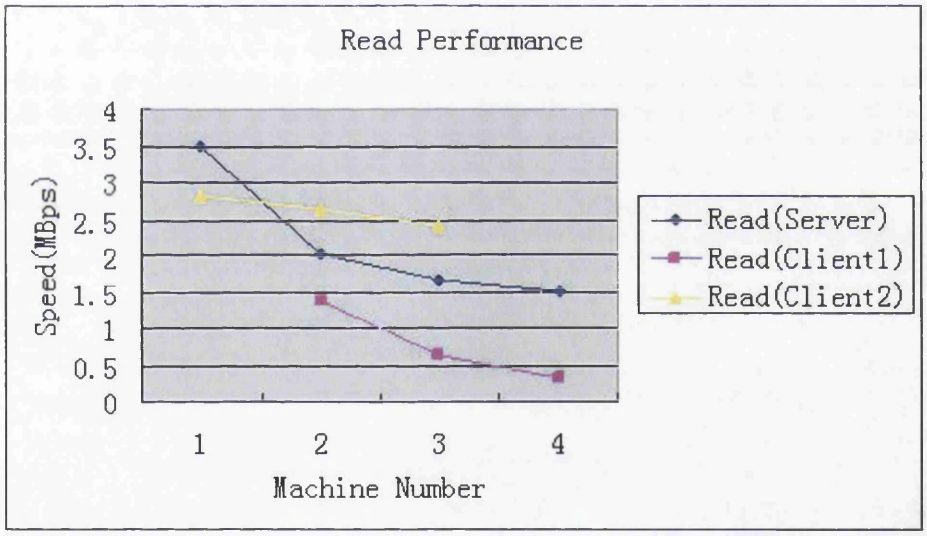

Figure 53

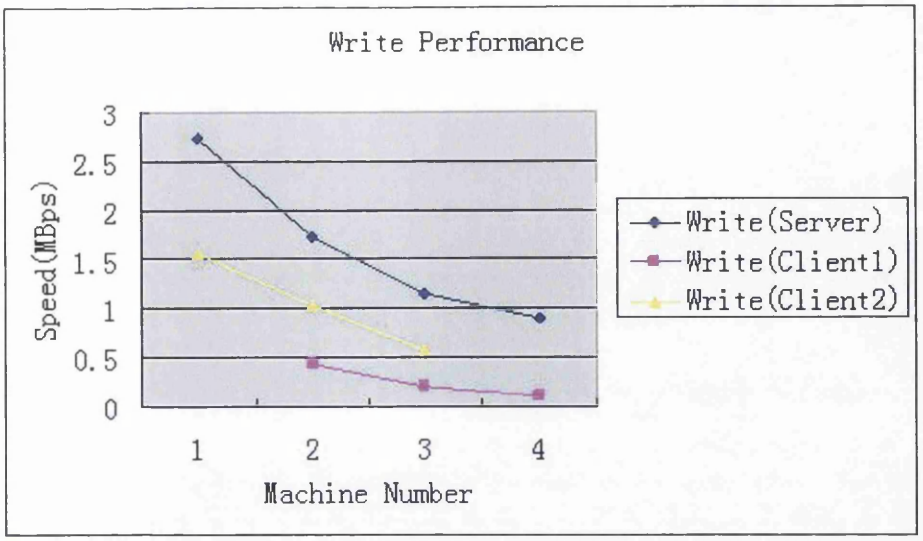

Figure 54

From these graphs, we see that the I/O performance drops severely when the number of machines increases. Especially when the server is also involved in the test, the I/O performance is extremely poor. The read/write speeds locally are over ten times faster than remote speeds in the worst case This is the reason for the poor performance of SGE and OpenPBS when running the I/O intensive job stream.

These results can be used to improve the scheduling performance of the Grid Scheduling Veneer. For instance, an I/O intensive job should be preferentially dispatched to a Condor site, if one is available.

8.3.2 The scheduling performance of the Grid Scheduling Veneer

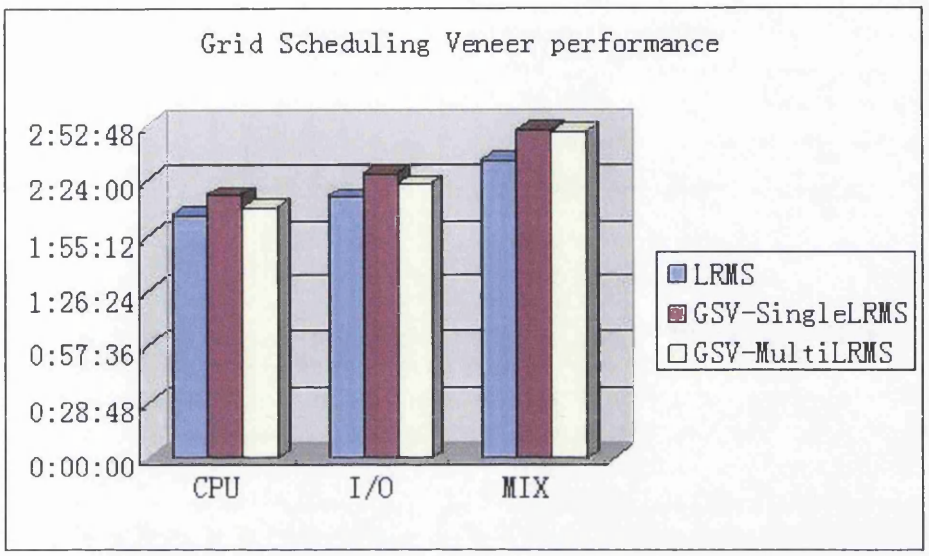

Figure 55

1) Overhead of Grid Scheduling Veneer

Comparing the stage1 and stage2 experiments results (Figure55), the Grid Scheduling Veneer introduces a few minutes overhead relative to submissions direct to the LRMS in the experiments. This is because the Grid Scheduling Veneer provides added value, such as description language translation, scheduling, and dispatching. Also, since the Grid Scheduling Veneer is an instance of a Grid/Web Service, it experiences the scheduling and communication delays typical of such services. From the experiment data, we find that about 74% of the time is spent on the initialization of the instance of the Grid Service and job dispatching, 8% is spent on the value added functionality, and 18% is spent on file transfer.

From the experiment in section 7.3.4, the overhead of the GSV is almost a constant value. In this experiment, the value is about 53 seconds. The results are shown in the following diagram.

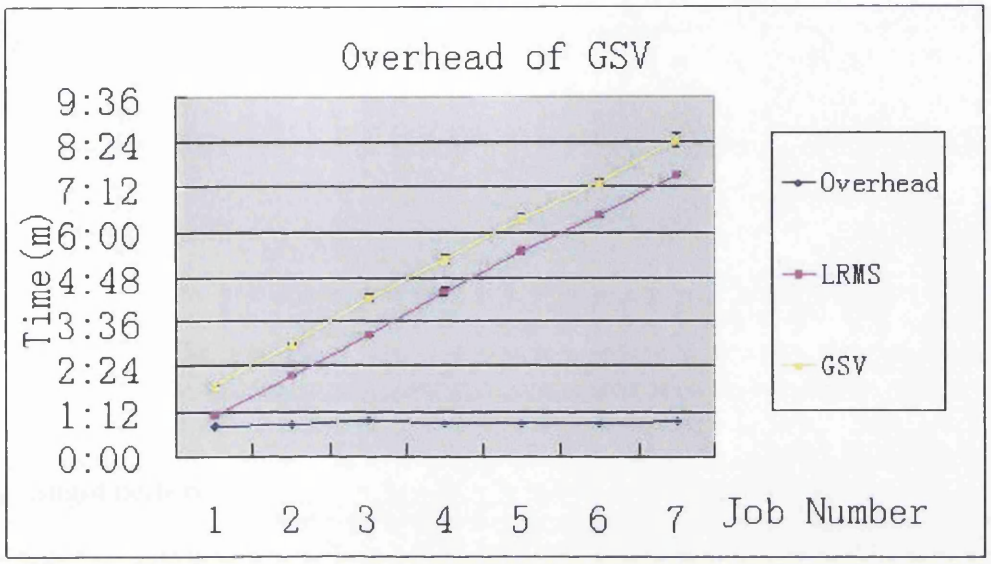

Figure 56

In order to prove that the prediction in section 7.3.4 is correct, we chose another job stream to validate the results. In this stream, the processing time of the GSV is greater than the execution time of single job in the LRMS. The results are shown in the following chart.

From the results, we can see that the overhead increases with job number. It agrees with the prediction. In addition, because the results also show that the overhead is nearly linear. Thus it also validated that the processing time of the GSV is a constant rather than a function related to the job's number in one batch.

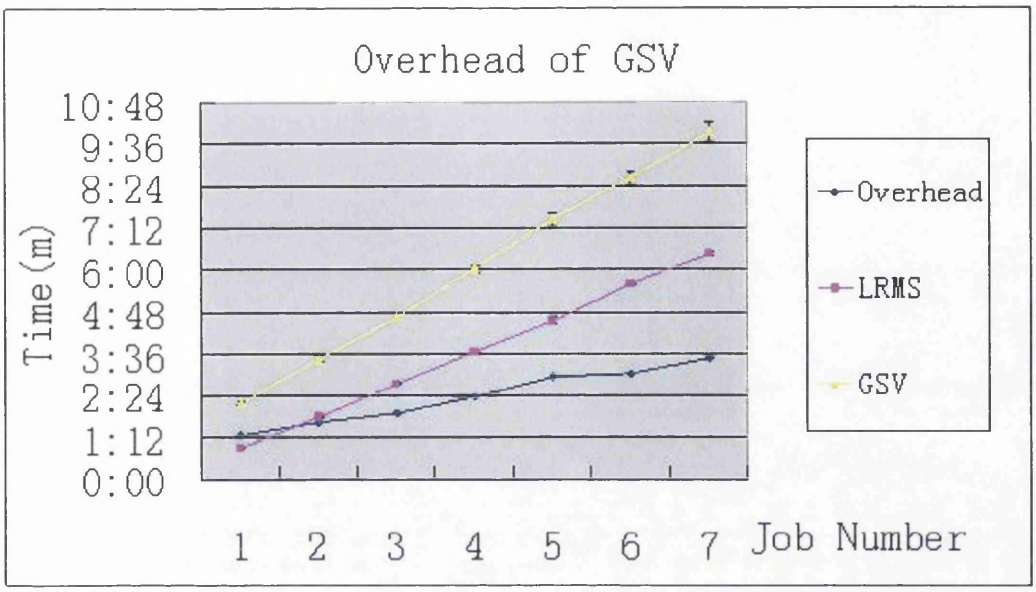

Figure 57

2) Slight performance improvement of Grid Scheduling Veneer

Figure 55 also shows a few performance improvements due to the Grid Scheduling Veneer. This is due to the scheduling algorithm implemented by the Grid Scheduling Veneer which balances the load and queue length of the execution sites. It avoids the situation that two high load jobs are dispatched to the same site where the overall execution time will be very long.

Chapter 9 - Conclusions and future work

After finishing the experiments described above, the Grid Scheduling Veneer has been tested for over one hundred hours to examine if it satisfies the design goals of the project.

The system goals partly fulfilled

According to the experiments, the Grid Scheduling Veneer achieves most of the system goals. Firstly, only one job description language – ERSL – is adopted in the Grid Scheduling Veneer and the experiments show that a description can be translated to the three LRMS-specific languages of our candidate LRMS's. Secondly, jobs are dispatched to execution sites chosen by the Grid Scheduling Veneer, instead of forcing the user to choose the site. In addition, the Grid Scheduling Veneer can periodically collect static and dynamic system information from the execution sites, and it is able to use this information to make dispatching decisions. Finally, the overhead of the GSV is linear and within a reasonable scope. The performance of the GSV is relatively better when dispatching the jobs to instances of all three types of LRMS.

Because the time of the project is only one year long, the GSV just includes some primary attributes of the three LRMS's rather than forming a superset of these attributes. As a result, this system goal is only partially achieved.

Future work

Although the project has finished, there are some issues that need to be addressed in the future.

1. System Emulation

In these three LRMS's, there are many valuable functions that have not been implemented in the Grid Scheduling Veneer. For example, interactive jobs can be submitted to OpenPBS and SGE but cannot be submitted to Condor. This kind of job is quite useful when a job needs to exchange some information with its submitter. The Grid Scheduling Veneer should emulate functions which are missing from one or more LRMS's.

2. More intelligent dispatching algorithm

Although the Grid Scheduling Veneer achieved some enhancement of overall performance, there is still room for improvement. From the experimental results, some exceptions need to be emphasized. For example, as shown in Figure 48, sometimes the execution time is extremely long. This is a shortcoming of the scheduling algorithm because it only considers two factors – Queue Size and System Load – to make its dispatching decision. Some job attributes such as job type and expected execution time should also be included as arguments to the scheduling function.

3. Adopts the newest research achievement

Because the scheduling issue is critical to the Grid system, many projects also pay lots of attention on it, especially in the GGF. The JSDL project will release

the first version Job Scheduling Description Language specification in this year. The GSV should fit itself in the framework of the JSDL as to make use of the pros of the JSDL such as flexible extensibility. By doing this, the GSV also can interoperate with other Grid applications that follow the JSDL specification easily.

References

1. Berman, F. Fox, G. and Hey, T. "The Grid: past, present, future" in Grid Computing: Making the Global Infrastructure a Reality (eds. F. Berman, G. Fox, and A. J. G. Hey) Wiley, 9-50.
2. D. de Roure, M. Baker, N. R. Jennings and N. Shadbolt (2003) "The evolution of the Grid" in Grid Computing: Making the Global Infrastructure a Reality (eds. F. Berman, G. Fox, and A. J. G. Hey) Wiley, 65-100.
3. The Globus Toolkit, http://www-unix.globus.org/toolkit/.
4. Foster, I. and Kesselman, C. (eds) (1999) The Grid: Blueprint for a New Computing Infrastructure. San Francisco, CA: Morgan Kaufmann.
5. Foster, I., Kesselman, C. and Tuecke, S. (2001), *The anatomy of the grid.* International Journal of Supercomputer Applications, 15(3).
6. The FAFNER, http://www.npac.syr.edu/factoring.html.
7. The Globus, http://www.globus.org/about.html.
8. Foster, I., Geisler, J., Nickless, W., Smith, W. and Tuecke, S. (1997) Software infrastructure for the I-WAY high performance distributed computing experiment. Proc. 5th IEEE Symposium on High Performance Distributed Computing, 1997, pp. 562–571.
9. Grimshaw, A. et al. (1997). The legion vision of a worldwide virtual computer. Communications of the ACM, 40(1), 39–45.

10. Ian Foster, Argonne National Laboratory, Argonne, Illinois, United States. "The Grid: A new infrastructure for 21st century science".

11. Tuecke, S., Czajkowski, K., Foster, I., Frey, J., Graham, S. and Kesselman, C. (2002) Grid Service Specifications, February 15, 2002, http://www.globus.org/ogsa.

12. The SOAP specification, W3C, http://www.w3.org/TR/soap/.

13. Kunszt, P., Guy, L. The Open Grid Services Architecture and data Grids. Grid Computing – Making the Global Infrastructure a Reality. 2003 John Wiley & Sons, Ltd.

14. S. Tuecke, K. Czajkowski, I. Foster, J. Frey etc. Open Grid Services Infrastructure (OGSI). http://www.ggf.org/ogsi-wg.

15. The Condor project, University of Wisconsin, USA, http://www.cs.wisc.edu/condor/.

16. The OpenPBS project, Altair Grid Technologies, http://www.openpbs.org.

17. The Grid Engine project, Sun Microsystems, http://gridengine.sunsource.net/.

18. The Apache XML Project, http://xml.apache.org/.

19. Schopf, J. (2002), A General Architecture for Scheduling on the Grid. Argonne National Laboratory ANL/MCS-P1000-10002.

20. The Globus project website: http://www-fp.globus.org/about/news/rd100.html.

21. The Condor-G project , http://www.cs.wisc.edu/condor/condorg/

22. Ann Chervenak, Ian Foster, Carl Kesselman, Charles Salisbury, Steven Tuecke, The Data Grid: Towards an Architecture for the Distributed Management and Analysis of Large Scientific Datasets (1999)

23. Grimshaw, A. Natrajan, A. HumphreyFrom, M. (2003) Legion to Avaki: the persistence of vision

24. Smarr, L. and Catlett, C. E. (1992) Metacomputing. Communications of the ACM, 35(6), 44–52.

25. S.Tuecke, K.Czaikowski, I.Foster, J. Frey et al. (2003) Open Grid Services Infrastructure (OGSI) Version 1.0, Globus Grid Forum

26. IBM Corporation. (1993) IBM Load Leveler: User's Guide. IBM Corporation.

27. Jackson, D., Snell, Q. and Clement, M. (2001) Core algorithms of the Maui scheduler. Proceedings of the 7th Workshop on Job Scheduling Strategies for Parallel Processing, 2001.

28. Cray Inc. (1997) Introducing NQE, Technical Report 2153 2.97, Cray Inc., Seattle, WA, 1997.

29. Jones, J. P. (1996) NAS Requirements Checklist for Job Queuing/Scheduling Software. NAS Technical Report NAS-96-003, April, 1996, http://www.nas.nasa.gov/Research/Reports/Techreports/1996/nas-96-003-abstract.html

30. Zhou, S. (1992) LSF: Load sharing in large-scale heterogeneous distributed systems. Proceedings of the Workshop on Cluster Computing, 1992.

31. Thain, T. Tannenbaum, T. Livny, M. (2003) Condor and the Grid. – Making the Global Infrastructure a Reality. 2003 John Wiley & Sons, Ltd.

32. Veridian Information Solutions, Inc. (2000) Portable Batch System Administrator Guide

33. Sun Microsystems, Inc. (2002) Sun Grid Engine 5.3 Administration and User's Guide

34. The Sun Data and Compute Grids Project, http://www.epcc.ed.ac.uk/sungrid/

35. Condor on WAN project, http://server11.infn.it/condor/condor-impl.html

36. OGSA-DAI project, http://www.ogsadai.org.uk/

37. OGSA-Grid project, http://sse.cs.ucl.ac.uk/UK-OGSA/

38. The New York Times, http://www.nytimes.com/2001/08/02/technology/02BLUE.html?pagewanted=print

39. The Legion CCA Project, https://www.cs.binghamton.edu/~gridweb/projects/legioncca.html

40. The Global Grid Forum, www.ggf.org

41. The Web Services Activity, http://www.w3.org/2002/ws/

42. A, Grimshaw. M, Lewis. A, Ferrari. J, Karpovich.(2000) Architectural Support for Extensibility and Autonomy in Wide-Area Distributed Object Systems Proceedings of the 2000 Network and Distributed System Security Symposium (NDSS2000), February 2000

43. H, Zhu. M. Siegel, S, Madnick(2001) Information Aggregation -A Value-added E-Service. The Proceedings of the International Conference on

Technology, Policy, and Innovation: Critical Infrastructures, The Netherlands, June 26-29, 2001.

44. ODDGenes Project, http://www.epcc.ed.ac.uk/oddgenes

45. The JSDL-WG Project, http://forge.gridforum.org/projects/jsdl-wg

46. The BRIDGES project, http://www.brc.dcs.gla.ac.uk/projects/bridges

47. M, Bayer. A, Campbell. D, Virdee.(2004) A GT3 based BLAST grid service for biomedical research. UK e-Science All Hands Meeting, 2003