Designing a resource broker for heterogeneous grids

SP&E

Srikumar Venugopal, Krishna Nadiminti, Hussein Gibbins
and Rajkumar Buyya*,†

Grid Computing and Distributed Systems Laboratory, *Department of Computer
Science and Software Engineering*, *The University of Melbourne*, *Australia*

SUMMARY

**Grids provide uniform access to aggregations of heterogeneous resources and services such as computers,
networks and storage owned by multiple organizations. However, such a dynamic environment poses
many challenges for application composition and deployment. In this paper, we present the design of the
Gridbus Grid resource broker that allows users to create applications and specify different objectives
through different interfaces without having to deal with the complexity of Grid infrastructure. We present
the unique requirements that motivated our design and discuss how these provide flexibility in extending
the functionality of the broker to support different low-level middlewares and user interfaces. We evaluate
the broker with different job profiles and Grid middleware and conclude with the lessons learnt from our
development experience. Copyright © 2007 John Wiley & Sons, Ltd.**

1. INTRODUCTION

Grids [1] enable collaboration through virtualization and sharing of resources and processes, owned,
managed and used by multiple organizations and individuals from across the globe. Originally born
out of the needs of high-end scientific investigations, this model of computing has come to be used in
a wide range of areas including scientific research, engineering, business, finance and manufacturing.
Grids consist of distributed heterogeneous resources such as high-end supercomputers, clusters,
storage repositories, databases and scientific instruments connected by high-speed networks, thereby
representing a wide variety of computing platforms and software systems.

---

*Correspondence to: Rajkumar Buyya, Grid Computing and Distributed Systems Laboratory, Department of Computer
  Science and Software Engineering, The University of Melbourne, Australia.
†E-mail: raj@csse.unimelb.edu.au

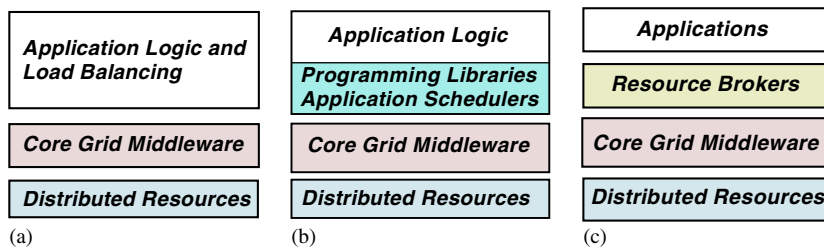| Application Logic and Load Balancing	Application Logic	Applications
	Programming Libraries Application Schedulers	Resource Brokers
Core Grid Middleware	Core Grid Middleware	Core Grid Middleware
Distributed Resources	Distributed Resources	Distributed Resources
(a)	(b)	(c)

Figure 1. Evolution of application development on Grids: (a) direct to core middleware (1995–); (b) through programming libraries (1999–); and (c) using user-level middleware (2000–).

The capabilities that need to be provided in order to realize a Grid environment include: uniform authentication and authorization; resource management and job submission; large-scale data management and transfer; and resource allocation and scheduling. Software tools and services that provide these capabilities are collectively called *Grid middleware*, which mediate between users and the underlying Grid fabric consisting of heterogeneous computing and storage resources connected by networks of varying capabilities. Figure 1 shows the evolution of Grid application development using Grid middleware.

The first generation of Grid middleware aimed to present a secure and standard method of invocation that abstracted the underlying heterogeneity of distributed resources. These *core grid middleware* such as Globus [2] and Legion [3] (Figure 1(a)) provided services for performing low-level Grid functions such as data access, job submission and authorization. Some of the early Grid applications directly invoked the functions presented by these *core middleware* through their application programming interfaces (APIs). However, they were still too complex and low level to become popular for general application development. These were followed by programming libraries such as NetSolve [4], Ninf [5], Cactus [6] and GrADS [7] (Figure 1(b)), which provided a software environment to create applications that accessed distributed services in a transparent fashion. While these libraries did a good job of shielding the developer from having to deal with varying Grid conditions, significant effort was still required to develop schedulers and task managers for each application. Projects such as AppLeS (Application Level Schedulers) [8] produced some of the early work in this regard. The next step was to move the scheduling algorithms to a generic framework that would also provide capabilities such as resource selection, job monitoring and data access to any application. Such frameworks are called *resource brokers* (Figure 1(c)) and some examples of these are Nimrod/G [9], AppLeS Parameter Sweep Template (APST) [10], Condor-G [11] and the Gridbus broker, which is the subject of this paper.

A previous publication [12] introduced the Gridbus broker as a scheduler for distributed data-intensive applications on global Grids. A case study using a data-intensive particle physics application was also presented to illustrate the use of the broker. However, this paper takes a detailed look at the architecture and design of the broker, and discusses how these were motivated by the requirements of working in a service-oriented, heterogeneous Grid environment where each user has his own quality-of-service (QoS) expectations to be satisfied. The Gridbus broker is then compared with related projects to highlight its uniqueness and the differences in function that have resulted from its design. Finally, the paper presents measurements that show the impact of the design on the performance of the components of the broker in different scenarios.

1.1. Resource brokers: challenges and requirements

The characteristics of Grids such as large number of services with multiple configurations, dynamic resource conditions and users with varying application requirements introduce unique challenges that are listed as follows:

Service heterogeneity: With the introduction of the OGSA [13], Grids have progressed from being aggregations of heterogeneous resources to collections of stateful services. These services can be grouped into several categories such as job submission and monitoring, information, data management and application deployment. Rapid developments in this field have meant that middleware itself changes frequently, and therefore service interface changes are the norm rather than the exception. While these interfaces are being standardized in fora such as the Open Grid Forum (OGF), a lot of Grid applications are being developed and deployed in enterprises and *academia* using existing non-standard interfaces. Therefore, due to this heterogeneity of service interfaces, supporting diverse service semantics is still a serious challenge.

Variety of application models: Grid applications tend to follow a variety of models such as bag-of-tasks, workflows, independent jobs and hosted application services. However, these are still required to interact with the same set of service interfaces. Enabling this interaction requires reconciliation between different application directives and constructs. Also, applications may invoke services in a variety of ways. A brokering system must avoid imposing constraints on applications as much as possible so as not to limit its own applicability.

Multiple user objectives: Applications and users may wish to satisfy different objectives at the same time. Some possible objectives include receiving results in the minimum possible time or within a set deadline, reducing the amount of data transfer and duplication, or ensuring minimum expense for an execution or minimum usage of the allocated quota of resources. Different tasks within an application may be associated with different objectives and different QoS requirements. A brokering system must, therefore, ensure that different scheduling strategies meeting different objectives can be employed whenever required.

Interface requirements: The interface presented to the user may take several forms. Many scientists are comfortable with traditional command-line tools and require that Grid tools should be command line and script-friendly as well. Web portals that allow users to invoke Grid and application capabilities within one interface have gained popularity in recent times as they enable the portability of working environments. Recent applications also seamlessly access Grid functions whenever required by invoking Grid/Web services or Grid middleware APIs. A resource broker should be able to support as many of these interfaces as possible in order to be useful to the largest community possible.

Infrastructural concerns: The quintessential properties of Grid environments such as the absence of administrative control over resources, dynamic system availability and high probabilities of failure have been described extensively in previous publications [14,15]. A resource broker has to be able to handle these properties while abstracting them as much as possible from the end-user. This is a significant challenge to developing any Grid middleware.

The following sections present the architecture, design and implementation of a Grid resource broker that takes into account the challenges mentioned so far in order to abstract the vagaries of the environment from the end-user.

2. ARCHITECTURE OF THE GRIDBUS BROKER

The architecture of the Gridbus broker is shown in Figure 2, and consists of three primary layers (interface, core and execution) with the database as a persistent backend throughout. This architecture isolates the logic (at the core) from interactions with both the user/application interfaces and remote Grid services. The components of the broker are grouped into the tiers based
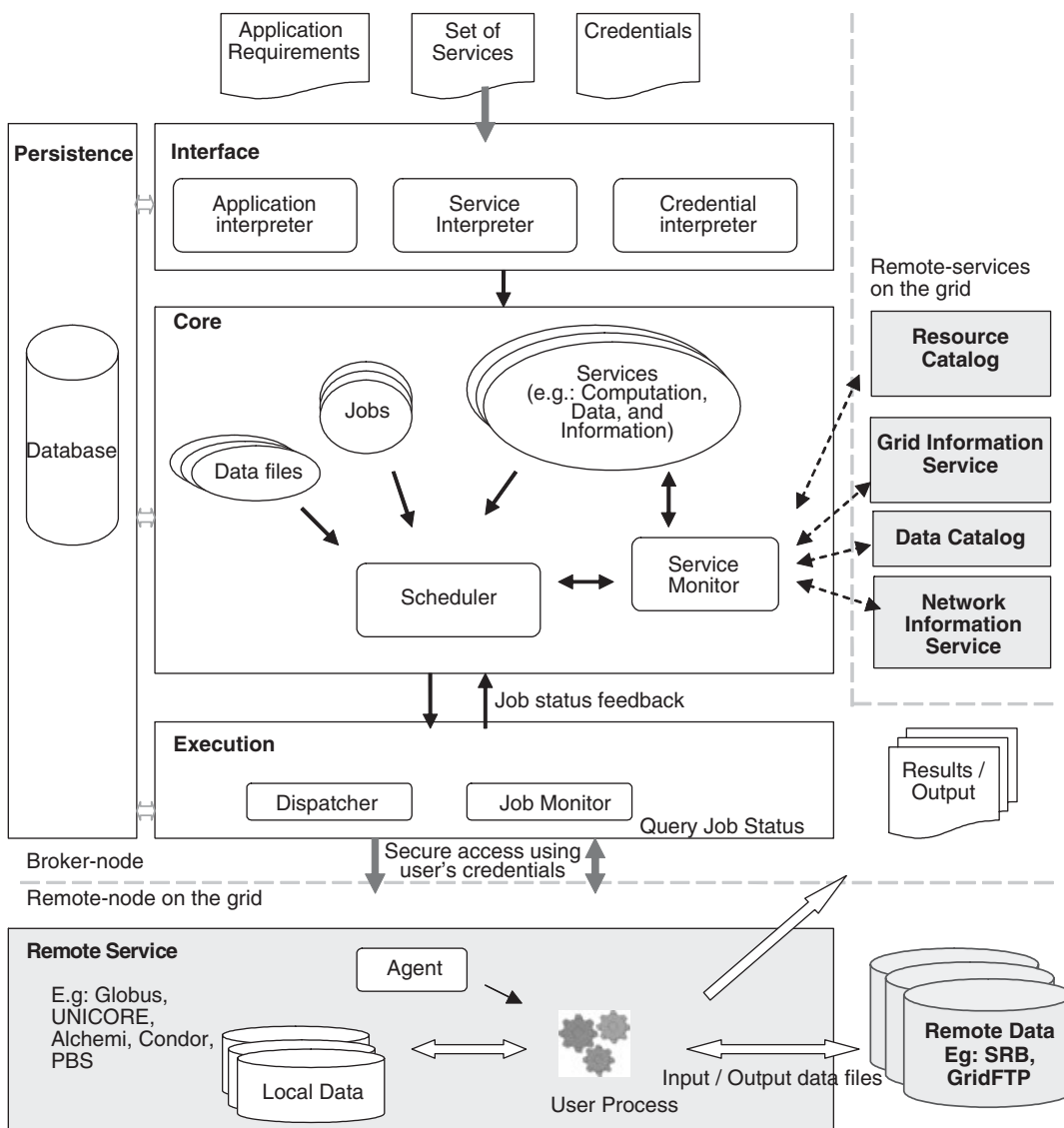

Figure 2. Gridbus broker architecture and its interaction with other Grid entities.

on the level of abstraction they provide from the underlying Grid resources. The overall flow of control is from top to bottom, while any events and exceptions that occur during execution are filtered by each tier from the bottom to the top. The tiers are described in detail in the following paragraphs.

2.1. Interface layer

Applications, Web portals and other such interfaces external to the broker interact with the components of the interface layer. The inputs from the external entities are translated by this layer to create the objects in the core layer. Three kinds of inputs are provided to the broker: a description of the application requirements, a set of services that can be utilized for executing the application and the set of credentials for accessing the services.

The application description provides details of the execution such as what is to be executed, the description of task inputs including remote data files (if required), the information about task outputs (if present) and the desired QoS. This description can be provided in one of the XML-based languages supported by the broker, or through the broker's APIs. Similarly, the set of services required for the user objectives can be provided through the APIs or as an XML-based service description file containing information such as service location, service type and specific details such as remote batch job submission systems for computational services. The services can also be discovered by the broker at runtime from remote information services such as the Grid Market Directory (GMD) [16] or Grid Index Information Service (GIIS) [17] among others. The list of credentials is provided in another file. File-based inputs are handled by the respective interpreters which convert the descriptions to entities within the broker. The application interpreter converts the application description file to task objects, while the service interpreter converts the service description to service objects. The credential interpreter similarly creates credential objects that are associated with a user and used by the broker to authenticate while invoking services (e.g. dispatching jobs on computational resources). These objects are described as a part of the core layer in the following section.

2.2. Core layer

This layer contains entities that represent the properties of the Grid infrastructure independent of the middleware and the functionality of the broker itself. Therefore, it abstracts the details of the actual interaction with the Grid resources performed by the execution layer. This interaction is driven by the decisions made by the functional components of the broker present in the core layer. These components can be broadly classified into two categories—*entities* and *workers*. This terminology is derived from standard business modelling processes [18].

Entities exist as information containers representing the properties, functions and instantaneous states of the various architectural elements that are proxies for the actual Grid entities and constructs involved in the execution. Therefore, entities are stored in the persistence database and are updated periodically. Workers represent the functionality of the broker, that is, they implement the actual logic and manipulate the entities in order to achieve the application objectives. Therefore, workers can be considered as active objects and the entities as passive objects. The next section (Section 3) takes a closer look at the entities and workers within the broker, accompanied by UML 2.0 [19] diagrams that illustrate the relationships between the components.

2.3. Execution layer

The actual task of dispatching the jobs is taken care of by the execution layer, which provides dispatchers for various middleware. These dispatchers create middleware-specific agents from the jobs and are executed on the remote resources. If there are data files associated with the job, then the agents request them from the data repositories that have been selected to access those files. During execution, the job monitor keeps track of the job status—whether a job is queued, executing, has finished successfully or has failed on the remote resource. On completion of job execution, the associated agent returns any results to the broker and provides debugging information.

2.4. Persistence subsystem

The persistence subsystem extends across the three layers described previously and maintains the state of the various entities within the broker. It is primarily used to interface with the database into which the state is stored at regular intervals. The persistence subsystem satisfies two purposes: it allows for recovery in case of unexpected failure of the broker and is also used as a medium of synchronization among the components in the broker.

3.   DESIGN OF THE GRIDBUS BROKER

This section presents the design of the Gridbus broker in detail. As mentioned before, the components in the broker are termed as entities or workers depending on whether they are passive or active objects. We will take a closer look at each of these, and later will explain how the design was driven by the requirements mentioned in Section 1.1.

3.1. Entities

3.1.1. *Application context and tasks*

An `ApplicationContext` represents the attributes of a user's application specification, such as the description of the application to be executed, one or more credentials for accessing the services for performing the execution, and the QoS requirements codifying the user's expectations of the execution, such as a deadline by which the execution must be completed. Figure 3 shows the placement of this object in relation to tasks and jobs. The `ApplicationContext` provides a single point for accessing a user's requirements.

A `Task` object represents the activities to be carried out in order to satisfy the user's requirements. Examples of activities include copying a file to the remote node, running the executable and storing the output in a remote repository. A `Task` consists of multiple `TaskCommands`, each of which encodes an activity, and which are carried out in sequence. Figure 3 illustrates three activities that inherit from the task command—the `CopyCommand` object, which represents a file transfer between two machines, the `ExecuteCommand` object, which specifies the executable and its arguments, and the `SubstituteCommand`, which specifies files within which variable values have to be substituted at runtime. The `TaskCommand` structure makes it possible to add more types of activities as and when there are requirements from applications.
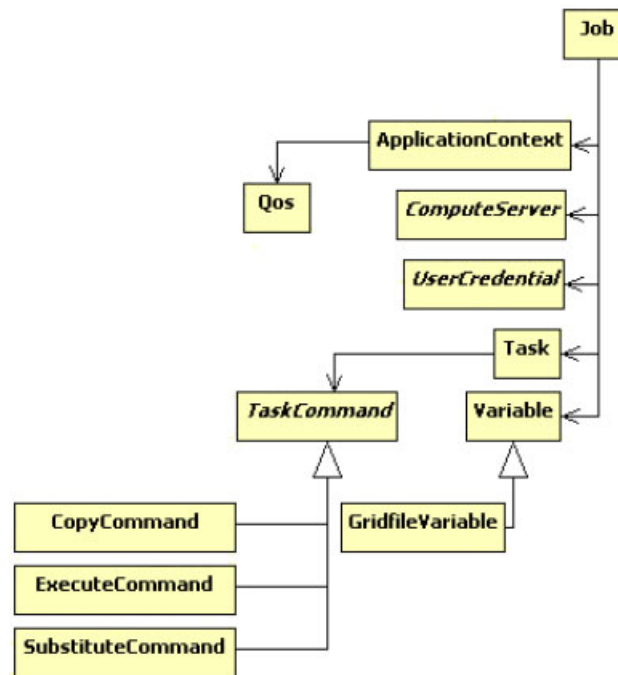
Figure 3. ApplicationContext, tasks and jobs.

Tasks are associated with input parameters encoded in `Variable` objects described later. The task specification provides the template for creating jobs which are the actual units of work sent to the remote Grid resources. That is, a task is an abstraction of the work performed for executing the application. The conversion of task to jobs is scenario dependent; for example, a parameter sweep task is converted into a set of jobs, while a single, independent task representing a simple application with a single function is converted into one job. The task structure within the broker is based on and extended from the task specification followed by Nimrod [20] for parameter sweep applications (PSAs). This task structure was chosen as it was generic, simple and could be mapped to any task description language. For example, the broker supports two different task specification languages: XPML (eXtended Parametric Modelling Language), an XML-based language created by us based on Nimrod's 'plan file' format for describing PSAs; and JSDL (Job Submission Description Language) [21], an OGF standard for describing independent batch job submissions.

3.1.2. Job

A `Job` represents an instantiation of the task at the remote node and is therefore associated with a single task object that describes its function. The structure of the Job object is shown in Figure 3. A job may be associated with one or more `Variable` objects which describe its input data. A `Variable` can be one of various types including integer, string and float, and is associated with a single value derived from its domain. New types of variables can be introduced into the broker by

extending the `Variable` class. For example, it was extended to create a `GridfileVariable` which describes a file or a data set that is stored on a repository that is accessible through any of the supported file transfer protocols.

A `Job` is also associated with a `JobWrapper` that represents the interface for translating the user task specification to create an agent that can be executed by the middleware running on the designated compute resource. Therefore, the `JobWrapper` is necessarily middleware specific and is associated with the execution layer. We will discuss more about the `JobWrapper` in relation to scheduling and job dispatching. Other than these, the job is associated with the `ApplicationContext` of its Task specification, a set of services to which it is allocated and a `UserCredential`.

As shown in Figure 4, a job passes through many stages in the course of its lifetime. A job is an input to the scheduler, which allocates it to a set of resources based on its requirements. The job's
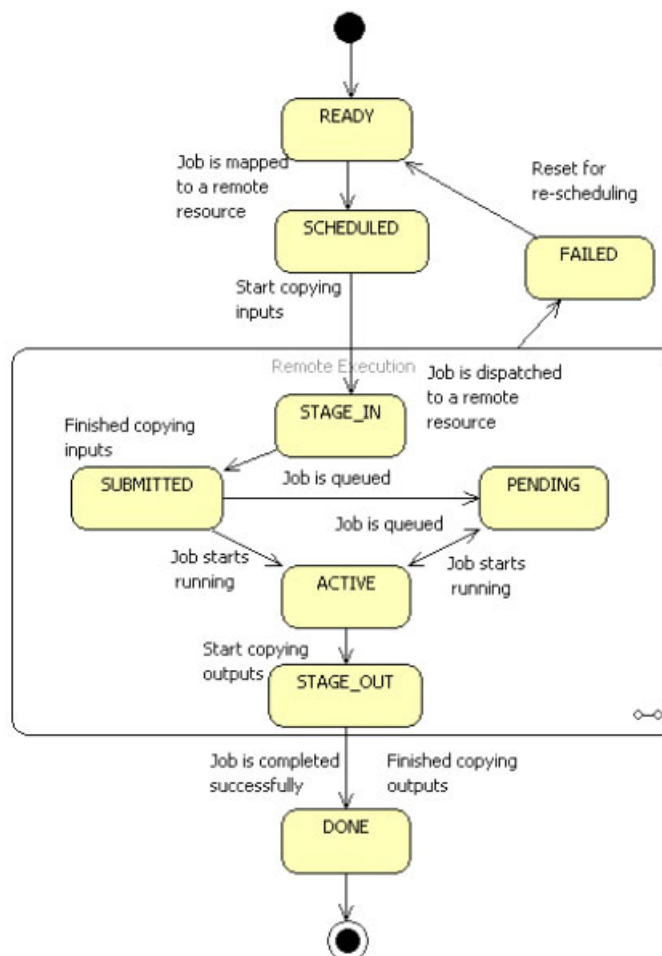

Figure 4. State transition diagram for a job.

status is then changed to SCHEDULED. During the STAGE_IN stage, input files and executables required for the job are staged to the remote resource. When this process is completed successfully and a handle is obtained, then a job is considered to be SUBMITTED. The job may be queued while waiting for an available processor and its state changes to PENDING. When the job starts its execution, it is considered ACTIVE. After the job has finished executing, it enters the STAGE_OUT stage, where its output files are transferred back to the broker. If all its outputs are received and are as expected by the task requirements, then the job is considered as 'DONE'. If one of the state transitions fails on the remote side or the job has completed on the remote side but has not produced the expected result files, then it is considered FAILED, and is reset and marked for re-scheduling. The job states within the broker were selected as a superset of the common batch job stages under various middleware and operating systems. This is to ensure that monitoring and reporting of the state of a job is uniform regardless of the execution environment. Also, the termination states DONE and FAILED are defined by the broker on the basis of its expectation of the job outcome (which is related to the user requirements). Therefore, even if the job has completed successfully on the remote machine but has not produced the expected output or has violated some constraints, then it may be considered as failed by the scheduler.

3.1.3. Services

It is possible to represent different types of services within the broker mirroring the variety of services that are available in Grids. These are represented in a hierarchy, shown in Figure 5, in which the parent class is an abstract `Service` whose attributes are the Service ID and the location of the service (hostname or Uniform Resource Indicator (URI)). The next level groups the services into categories depending on the type of the service. The `ComputeServer` object represents a computational resource with properties such as architecture, operating system and
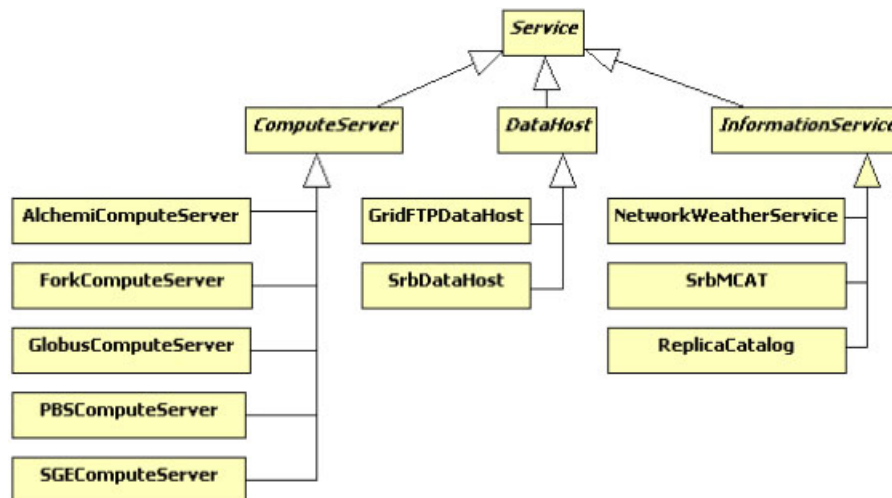

Figure 5. Service object hierarchy.

available job submission systems. `DataHost` objects describe storage repositories and the details of the data files stored within these. These details include attributes such as the location, size and the protocol used to access the files in the repository. Services that provide meta-information such as resource information services, data catalogues and market directories are represented by `InformationService` objects. `InformationServices` are categorized depending on the type of information they serve. For example, `ReplicaCatalog` services such as Globus Replica Catalogue and SRB MCAT provide information on different copies of required data sets that are stored on distributed repositories. Information about network properties is gathered from the `NetworkInformationService`. For each pair of resources, this information is stored in a `NetworkLink` object.

The lowest child classes in the hierarchy represent specific implementations of these services that provide the functions of associated middleware. For example, interaction with a computational resource running Globus middleware is implemented in the `GlobusComputeServer` class that extends the abstract `ComputeServer`. This interaction, which utilizes the Java Commodity Grids (Java CoG) Kit [22] in the Execution Layer, is kept isolated from the rest of the broker, to which the `ComputeServer` class is presented as a single compute resource. In production Grids, it is more likely that the remote resource may be a cluster running some sort of a batch job management system such as Portable Batch Scheduler (PBS) [23] or Sun Grid Engine (SGE) [24]. These job queueing systems may be accessible either through Globus job managers or directly using command line utilities. Many of these systems are set up to have queues with different priorities, number of available slots and limits on the duration of jobs. The broker supports such systems by providing facilities for scheduling jobs to queues based either on the user's specification or on the length of the job and the availability of slots. Similar to Globus, the abstract `ComputeServer` has also been implemented for different middleware and job managers such as Alchemi [25], Unicore [26], PBS, SGE, Condor [27] and XGrid [28]. In the case of the last four, the job management systems are invoked through an SSH (Secure Shell) connection.

Interaction with remote storage resources is implemented through the `DataHost` abstraction. Specific implementations of this abstraction provide specific functions of storage middleware. As an example, Figure 6 shows the list of methods available in the `SrbDataHost` class that implement functionality specific to a storage resource broker (SRB), such as zones and SRB authentication. Similarly, the `DataHost` has been implemented for providing access to data stores Grid enabled by GridFTP.

3.1.4. Credentials

The Gridbus broker defines the `UserCredential` object that represents an authentication token to access remote services. The base `UserCredential` object has been extended to realize the different types of credentials that are accepted by different middleware. For example, the `LocalProxyCredential` represents the Globus GSI (Grid Security Infrastructure) X.509 proxy object created on the client side. On the other hand, a `SimpleCredential` is used for accessing resources that require a username and a password for authentication, such as those enabled through SSH.

The set of credentials available to the user is associated with his/her `ApplicationContext`. Even though the credentials are passive objects, to ensure the security of users' credentials, these
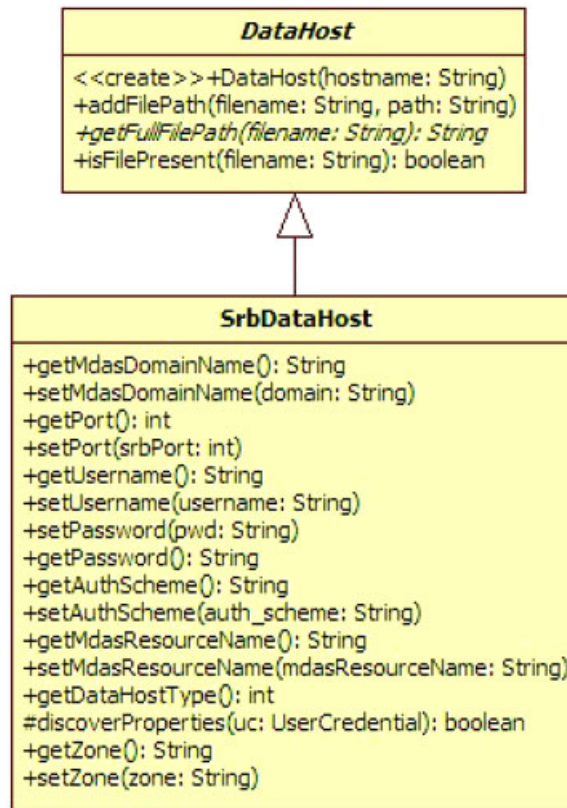
Figure 6. SrbDataHost as an instance of DataHost.

are transient and are not saved in the persistence storage. This also means that the user has to provide the broker with a fresh set of credentials when recovering from a previously paused/failed run of a Grid application. Figure 7 shows the various types of credentials supported by the broker.

3.2. Workers

3.2.1. Broker

The Broker object is the first component to be initialized and is a container for the other objects in the broker. It is responsible for managing the lifecycle of the other active objects—Scheduler, JobMonitor and ServiceMonitor—from start-up to shutdown. It is also a front end to the persistent storage and, therefore, maintains the overall state of the broker by saving the state of various passive entities. The broker and its associations with other entities within the broker are shown in Figure 8. The BrokerStorage class provides the methods for the other active objects
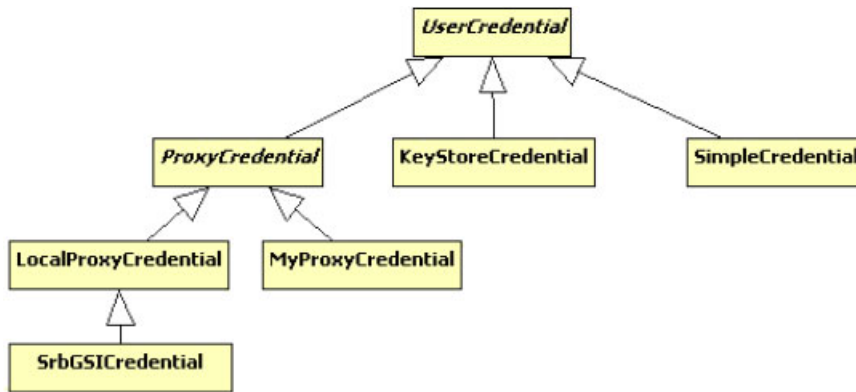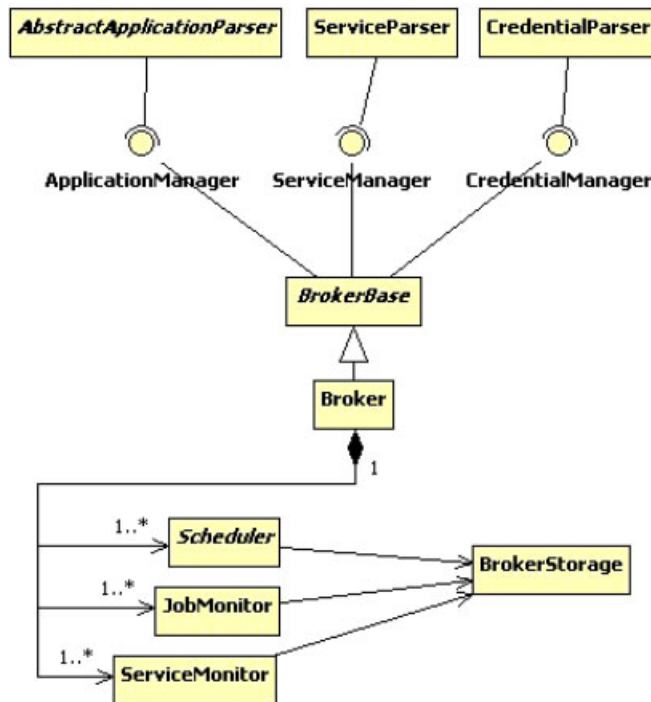
Figure 7. Credentials supported by the broker.

Figure 8. The broker and its associated entities.

to interact with the persistent database. The `Parser` objects convert the various input specifications to the entities within the broker and store them in the database.

The broker can be started as a single-user system on the command-line, a remotely hosted Web service compliant with the WSRF (Web Service Resource Framework) [29] standard or embedded

within another Java application. The WSRF interface allows the isolation of the broker functionality from the hosting application. A common usage scenario for this mode of operation is a Web portal that offers a front-end to e-Science applications. The WSRF interface for the broker is designed as an adaptor around the `Broker` component. The `WSRFBroker` is a wrapper for the broker class which parses the SOAP messages and delegates all the invocation to the broker internally.

3.2.2. ServiceMonitor

The `ServiceMonitor` component is responsible for maintaining the up-to-date view of the Grid resource status within the broker. It periodically checks the availability of the specified remote services and discovers new services which may have become available. It is able to retrieve information from various `Service` objects about the grid environment. The information retrieved is stored into the broker's database, from where it is retrieved by the scheduler for making decisions.

The `ServiceMonitor`, shown in Figure 9, is implemented such that it is independent of both the actual middleware details and the difference between the various types of services. The abstract `Service` entity provides a `discoverProperties` operation which is implemented by all services and is invoked by the `ServiceMonitor` to instantiate the properties of this entity. This operation performs a search for attributes that are specific to the service and middleware type. If the operation is successful, the values that are so retrieved are set within the `Service` object
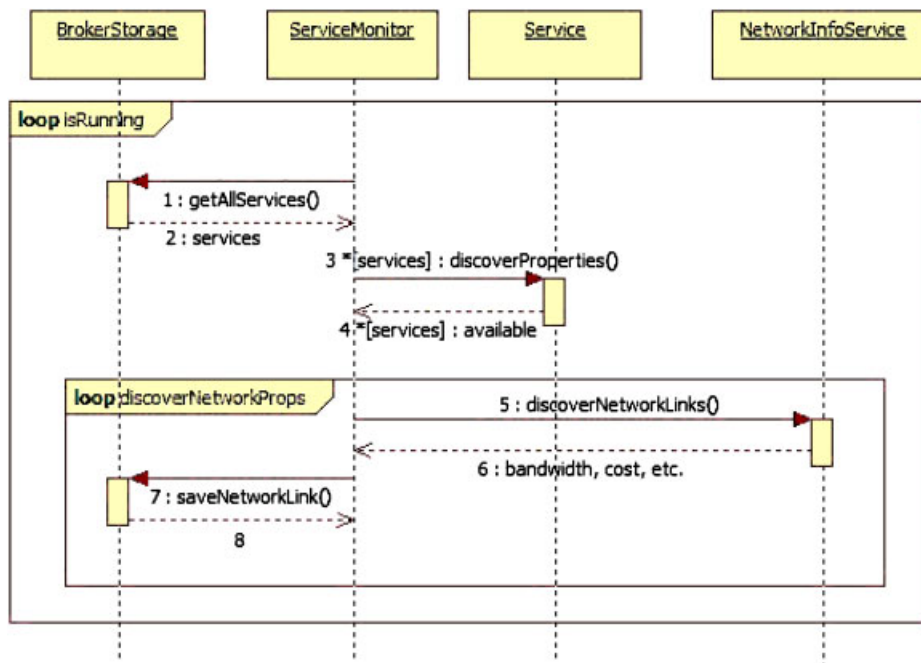

Figure 9. The service monitor and its operations.

and the `ServiceMonitor` is notified that the service is available. For example, for a Globus computational resource, the operation first checks if the credentials provided by the user can be used to authenticate to the remote resource. After this, it queries the Monitoring and Discovery Service (MDS) on the resource in order to obtain the values of attributes such as architecture, operating system, CPU and available memory. If a job manager such as PBS or SGE is available, then the next step is to query the available queues and their attributes (slots or wall-time constraints). For a data service, `discoverProperties` may query the amount of available storage and the data collections accessible by the user. In this manner, each server can define its own methods for discovering specific attributes.

Another function of the `ServiceMonitor` is to discover and refresh the properties of the network links between different services. As shown in Figure 9, this is performed through the `discoverNetworkLinks` operation, which refreshes the data stored in `NetworkLink` objects using the `NetworkInformationService`.

3.2.3.   Scheduler

The scheduling component is designed as two separate components: the `Scheduler` and the `Dispatcher`. The `Scheduler` matches the jobs individually to the services and also decides the order of execution of the jobs on the resources. Figure 10 shows the basic sequence of operations that is performed by the `Scheduler`. The `Scheduler` gets the list of ready jobs from the persistent storage and, depending on the strategy and the application requirements, a list of appropriate services. The mapping is an assignment of a job to appropriate computational and data services.
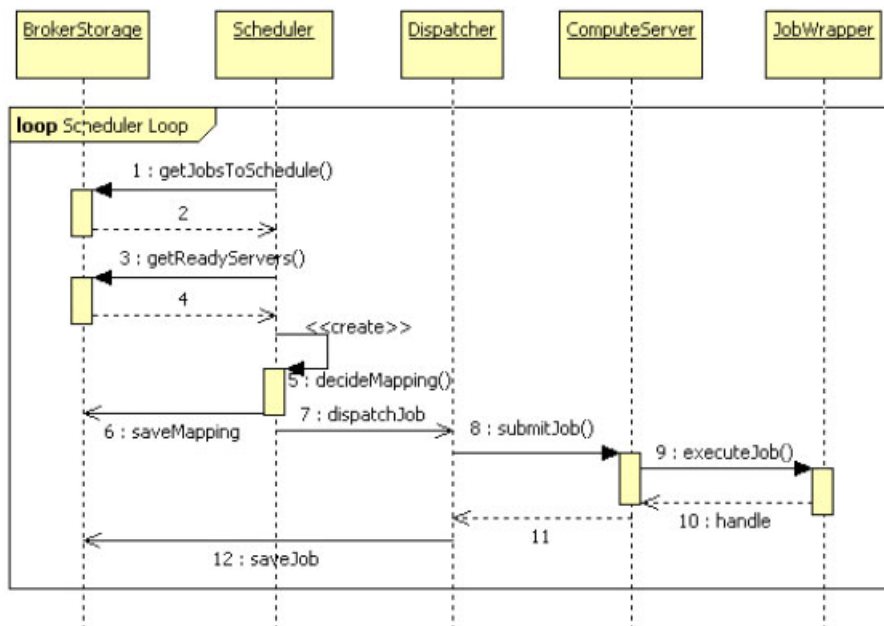

Figure 10. Schedule sequence diagram.

The job can specify minimum requirements for a qualifying resource. The `Scheduler` takes this into account and uses appropriate heuristics (based on user selection) to generate the mapping. At the very least, the job has to be mapped to a compute resource (or a `ComputeServer` object) where it is to be executed. If the application specifies data files to be processed, then the mapping also includes the assignment of `DataHost` objects from which each of the files should be accessed. At the same time, the state of the job is changed from READY to SCHEDULED.

The `Dispatcher` created by the `Scheduler` to dispatch a job creates a `JobWrapper` object depending on the type of `ComputeServer` selected for the job. The `JobWrapper` is a short-lived object which translates the abstract job to execute on the target middleware and architecture. For example, the `Dispatcher` would instantiate a `GlobusJobWrapper` object for a `GlobusComputeServer`, which translates the `TaskCommands` to invocations specific to the target architecture and Globus 2.x. It then performs job submission in accordance with the protocols followed by Globus.

Prior to the actual submission of the job, the files required for the job, such as input files and executables (if any), are copied on to the remote resource. The files can be transferred on to the resource by the broker (push model) or can be requested and copied by the remote resource (pull model). This allows a great deal of flexibility in implementing transfer modes, three of which are illustrated in Figure 11. For example, many Grid resources are behind firewalls that prohibit any connections to any port on the resource. In this case, using the pull model has the advantage that a transfer program on the resource can make an outbound connection through the firewall (Figure 11(b)). Another advantage of the pull model is that the source of the files can be a file server that is separate from the resource on which the broker is running, thereby supporting a scenario in which the broker is behind a firewall as well (Figure 11(c)). The resource manager is a middleware component that is able to send and receive messages from the outside world through the firewall. During this process, the job state is changed to STAGE_IN.

We illustrate this capability using an example of submitting a job and its associated data to a resource that is Grid-enabled using Globus 2.4 and located behind a firewall. The broker transfers the file to a GridFTP server and embeds the file location within the job, which is then submitted to the Grid resource. The job requests the file from the given location, thereby pulling it over to the resource. In case of Globus 4.0, the file location can be specified as a directive within the job's Resource Specification Language (RSL) [30], in which case the Globus 4 resource manager itself initiates file transfer.
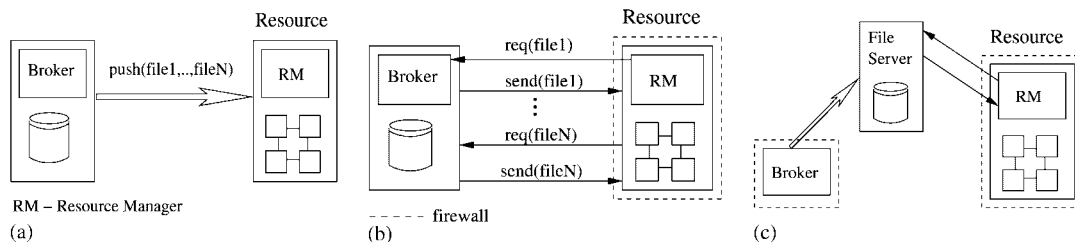

Figure 11. File transfer modes supported by the broker: (a) push model of file transfer; (b) pull model of file transfer; and (c) a file transfer using an intermediate server.

The `JobWrapper` submits the prepared job to the remote resource manager and waits for confirmation of acceptance. This is obtained as a remote handle to the job that uniquely identifies the job at the resource. The job state is changed to SUBMITTED and the job along with the handle is saved back to the persistent storage. The number of available slots within the `ComputeServer` is decremented to reflect the submission and the server also saved to the persistent storage. If the handle is not received, then the job submission is considered as FAILED and the job is marked for re-scheduling.

The `Dispatcher` is also able to follow a two-phase commit protocol for job submission as described by Czajkowski *et al.* [31]. In the first phase, the stage-in of the files and the job submission are performed, and the dispatcher waits for an 'agreement' message from the remote resource in the form of the handle. After the handle is received, the dispatcher sends a 'commit' message to the remote resource, which proceeds with the job submission and returns an acknowledgement. The job state is changed to SUBMITTED only after the acknowledgement is received. For resources running middleware that do not support the two-phase protocol, the receipt of remote handle is considered as acknowledgement of submission. In this case, if there is a network failure before the handle is received at the broker, the job will be marked as failed though it may have started execution at the remote node. In the two-phase protocol, the job is not processed by the Grid resource until the broker sends a commit message. Thus, the two-phase commits ensure that job submission is carried out only once even in the face of network problems.

Prior to passing control to the `JobWrapper`, the `Dispatcher` selects a credential from the set of `UserCredentials` and binds it to the job. This decision is based on the type of the middleware and the type of credential. Alternatively, a user may explicitly specify the mapping of credentials to resources. This feature aids a user to seamlessly run jobs on different types of middleware, or even to use different credentials for the same type of middleware, at the same time.

3.2.4.   *JobMonitor*

The `JobMonitor` object associated with the `ComputeServer` keeps track of a job's progress after submission to that remote resource. As shown in Figure 12, the `JobMonitor` periodically (the default is 30 s) requests the list of jobs that are in the SUBMITTED state on a particular resource from the persistent storage. It uses the remote handle to query the status of the job using middleware-specific functionality. The query is a blocking call, and is therefore provided with a timeout period after which the `JobMonitor` cancels the query to proceed to the next job. Failure to contact the job on the remote resource is not considered as a failure of the job immediately. The `JobMonitor` tries to contact the job again for a set number of times before giving up and marking the job as failed.

It is possible to implement `JobListeners` interfaces that will receive events from the JobMonitor when a job status is changed. The listeners can be entities within the broker, such as an event-driven scheduler, or outside the broker, such as an applet within an application Web portal that has been integrated within the Gridbus broker [32].

3.3.   Meeting challenges in the design

The primary aim of the Gridbus broker is to provide a generic resource discovery and scheduling framework that abstracts the heterogeneous and dynamic nature of the Grid infrastructure and allows
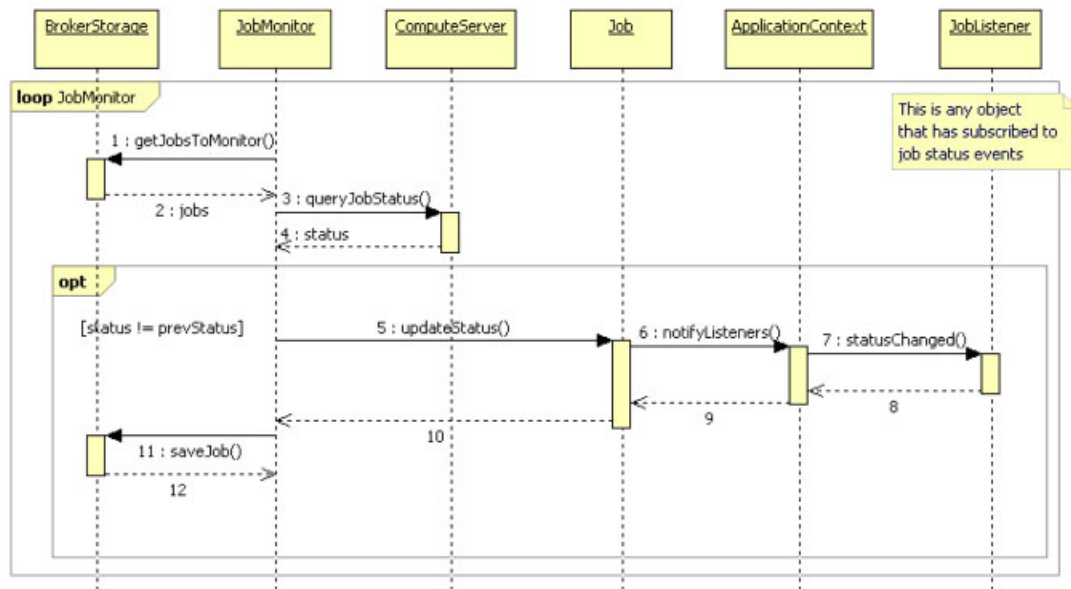
Figure 12. Job monitoring sequence diagram.

users to achieve different objectives. In Section 1.1, some of the challenges in achieving this aim are outlined. These challenges provide the requirements against which the broker is designed. The broker meets these requirements in the manner outlined as follows.

3.3.1. Service heterogeneity

The Gridbus broker tackles the problem of heterogeneous Grid resources and service interfaces by adopting the principle of *minimal assumptions*. This means that, throughout the broker, there are as few assumptions as possible about the nature of the environment in which it operates. The relationship between the objects in the broker is generic and independent of interaction models followed by any middleware. The broker, therefore, does not impose a particular configuration requirement on Grid resources and is thus able to use as many resources as possible. For example, a resource running any Unix-based operating system and with one of the supported middleware operational would be immediately useable by the broker as the latter requires only a POSIX-compliant shell environment that is standard on such machines.

The three-layer architecture of the broker also helps in maintaining this independence. For example, the assignment of jobs to resources is performed by the scheduler, which, as a component of the core layer, has a middleware-independent view of the resources, while the dispatcher dispatches the jobs to the resources. However, the actual interface with the remote resource happens through the middleware-specific `JobWrapper`, which is a part of the execution layer.

The broker can also be made to interface with any new service or middleware by extending the appropriate classes. For example, support for a new middleware can be added by extending the abstract `ComputeServer` and `JobWrapper` classes. The scheduler will then be able to

immediately utilize any resource running that middleware. The interaction with the remote middleware is also independent of the rest of broker. Using this method, the broker has been extended to support a wide variety of both computational and data Grid middleware such as Globus, PBS and SRB among others [33,34]. With the support for Alchemi, which is a .NET-based desktop Grid computing framework for Windows platforms, and XGrid, a similar framework for Mac OS X, the Gridbus broker can already schedule jobs across almost all of the platforms in use today. Similarly, new information sources and data repositories can be supported by extending the `InformationService` and the `DataHost` classes, respectively.

3.3.2. Support for different application models and user interfaces

The components within the broker are designed to be modular and there is a clean separation on the basis of functionality. Particularly, the division between workers and entities clearly delineates responsibilities among the components of the broker. Since the passive entities are just holders of information, the logic within the workers can be changed without affecting the former. It is also possible to introduce new workers that either use or extend the existing entities in different ways or introduce new entities of their own *without structural or design changes to the rest of the broker*. This loose coupling between components provides a lot of flexibility and enables the realization of different applications and system models [35].

The components within the interface layer convert the application, service and credential descriptions to broker entities and store them into the persistence database. Thus, it is possible to support any form of description by mapping it to the entities within the broker. At present, the broker supports the XML-based description of parameter sweep and bag-of-task applications and has a set of XML-based files for describing services and credentials. These inputs can also be provided directly to the broker through its APIs. The APIs allow direct manipulation of the entities in the broker and thus can be used to realize different application models.

The same mechanisms that allow the creation of application models also enable the creation of different user interfaces for the broker. The broker can be interfaced through command line, desktop clients or Web portals. It is also possible to talk to the broker's persistence database directly through its APIs. This enables any component to retrieve information about the broker entities through normal SQL (Structured Query Language) queries. This ability is useful in scenarios such as an application Web portal requiring information about an ongoing execution through a broker installed on a machine different from the portal server.

The WSRF Web service interface enables multi-user multi-application scenarios, where the broker is hosted as an isolated service that is accessible by Web portals using the WSRF Broker Client APIs. The separation between Web application and broker service reduces the memory and CPU usage in the Web server hosting the application, and increases performance and scalability. Moreover, the reliability of the application is also enhanced as failures in the broker will not affect the application server, and end users could still have the application Web portal working while the broker service recovers and resumes execution of the grid application.

3.3.3. Realization of different user objectives

The broker allows for schedulers to be plugged in, and hence is able to support new scheduling algorithms and policies. This enables the broker to adapt to new user requirements and

objectives. The separation of the dispatch component from the scheduling provides a lot of flexibility for implementing the scheduling logic. These two components are also designed to be independent of the resource and middleware details. However, a developer can still choose to create schedulers that may require certain middleware-dependent services such as resource information services.

One of the main requirements of the design was the ability to execute generic data-oriented applications that may require access to one or more distributed data sets at the same time. Data services, represented by `DataHosts`, have the same level of importance as `ComputeServers`. Location of available replicas of a data set can be gathered by querying the appropriate replica catalogs that are represented as `InformationServices`. Information about current network conditions, such as available bandwidth, is important when large data sets are to be transferred. This is available through the `NetworkLinks` data structure, as are properties such as pricing, classes of service and availability. `CopyCommands` in the task allow the application to perform third-party (not involving the broker machine) point-to-point data transfers on the Grid. The application interpreter allows users to specify data sets as input parameters to their applications. In this case, the `ServiceMonitor` will discover all the data repositories that host these data sets (through implementations of data catalog information services) and the `Scheduler` will select one of the repositories for accessing the data sets.

The Gridbus broker has been designed from the ground up to support the computational economy paradigm [36], and assigns costs to various services including computation, data storage and information services. The default scheduling policy uses these parameters to decide on an optimal mapping strategy to schedule jobs on resources. The design also allows the ability to plug-in a market directory that offers information about various priced services, and a Grid bank [37] that manages users' credit in a Grid market. The `ServiceMonitor` is able to periodically refresh pricing information from market information services and therefore, provides the ability to keep up with dynamic pricing and resource conditions in a volatile Grid economy. More importantly, the combination of all these features enables the broker to satisfy different requirements such as selecting data repositories for accessing data sets on the basis of price and/or performance and selecting network links offering a particular class of service [38].

The Gridbus broker is also able to support users who would like to use resources spread across multiple VOs by natively managing multiple credentials for multiple resources. This ability also allows users to access legacy data repositories that are not 'Grid enabled' and follow their own authentication mechanism, rather than Grid proxies.

3.3.4. Infrastructure

Grid environments are dynamic in nature. As a result, transient behaviour is a feature not only of the resources but also of the middleware itself. The broker design considers various types of failures that may occur either on a remote resource to which a job is submitted or on the broker machine itself during various stages of its operation. Remote failures include failure during job submission, execution and monitoring, or retrieving the outputs. These could be due to various reasons, such as incorrect configuration of the remote resource, incorrect job descriptions, network problems, unavailable data files or usage policy restrictions on the resource. Local failures include unexpected system crashes which lead to abrupt termination of the broker, unavailability of local input files and invalid input parameters.

The broker applies different fault recovery methods in different cases of remote failure. A job is not considered completed until it is determined that each of its constituent task activities has successfully exited. Thus, even if the middleware on the remote resource has signalled a successful completion, the `JobMonitor` checks to see if the job has generated the required results and only then it is deemed successful. The `JobMonitor` may not be able to contact an executing job in case of transient network conditions. In such cases, the `JobMonitor` polls the job a set number of times, and, if it is able to re-establish contact, then checks the job's current status. In all other cases of job failure, the job is rescheduled on another available resource.

The persistence system provides insurance against failure of the broker itself. At any point in the execution, the state of the broker is completely described by the contents of the entities and, therefore, only these need to be stored within the persistence database. Any change in the status of a job or a service, as discovered by the respective monitors, is immediately written back to the database so that the latter reflects the true state of the broker. Workers have no state of their own and, hence, can be resumed from the point at which they failed by reading the entities from the persistent storage.

Frequently, resource discovery and monitoring services on Grid resources are not configured properly. This means that information about a Grid resource may be unavailable, or stale, even if available. The Gridbus broker gets around this problem by not assuming the availability of a working information service. For example, in the case of a Globus resource, if the MDS service is not available but yet the resource is reachable, then the resource is considered for job submission. Also, the broker tracks variations in resource availability by monitoring resource performance locally. Nothing is required to be installed at the remote resources, and dependencies on metrics provided by the middleware are avoided by default. Thus, the broker is able to compare resources in a heterogeneous environment based on metrics that are independent of middleware and relative to the requirements of the current execution. However, it is to be noted that there is no mechanism in the broker to inhibit usage of external performance monitors and other services if required.

4.   RELATED WORK

The challenges presented in Section 1.1 have motivated the development of a large number of Grid resource brokering and application deployment systems. Examples of such systems are Nimrod/G, Condor-G, APST, EU-DataGrid Workload Management System (WMS, succeeded by the gLite [39]), GridWay [40], DI-GRUBER [41], SPHINX [42] and GridLab Grid Resource Management System [43] among others. In this section, however, the first four are chosen for detailed comparison against the Gridbus broker as their objectives and approach are similar to that of the broker. These are compared with the broker against the manner in which they handle the challenges outlined previously.

4.1.   Nimrod/G

Nimrod/G [9,44] is a tool for automated scheduling and execution of PSAs on Grids. It provides a declarative parametric modelling language through which the task specifications can be provided for an 'experiment' or execution of an application. Scheduling within Nimrod/G follows an economic model in which the resources have costs associated with them and the users have to expend their

budgets in order to execute their jobs on the resources. The user can also specify QoS requirements such as a deadline for finishing the experiment and an option for choosing between a faster yet more expensive execution vis-a-vis a slower but cheaper process. Architecture-wise, Nimrod/G consists of a Task Farming Engine (TFE) for managing an execution, a scheduler that talks to various information services and decides on resource allocations, and a dispatcher that creates agents and sends them to remote nodes for execution. An agent can manage more than one job at a remote site.

Nimrod/G works with UNIX-based resources enabled through Globus, Legion, PBS, Condor, SGE and others. It has the ability to specify data transfers between any two machines that may be running different Grid middleware. It takes into account different patterns of data usage, such as pipelined data between sequential processes to schedule jobs and to make better use of storage on resources [45]. Nimrod/G proposed the Grid economy paradigm and has implementations of four algorithms [46]—*time optimization*, *cost optimization*, *cost–time optimization* and *conservative time optimization*—for scheduling parameter sweep computationally intensive applications. It also implements a number of other scheduling algorithms from APST, leveraging services such as the Network Weather Service (NWS). Nimrod/G also provides additional interfaces that expose the functionality through Web Services and Web portals. Moreover, tools like Nimrod/O [47] allow users to specify complex search and optimization algorithms.

4.2. AppLeS parameter sweep template (APST)

APST [10] is an environment for scheduling and deploying large-scale PSAs on Grid platforms. APST provides mechanisms for deploying applications on different Grid middleware and schedulers that take into account PSAs with data requirements. APST consists of two processes: the daemon, which deploys and manages applications, and the client, which is a console for the users to enter their input. The input is XML-based and no modification of the application is required for it to be deployed on Grid resources. The APST scheduler allocates resources based on several parameters, including predictions of resource performance, expected network bandwidths and historical data. Examples of scheduling strategies include algorithms that take into account PSAs with shared input files [48] and Divisible Load Scheduling-based algorithms [49]. The scheduler uses a data manager and a compute manager to deploy and monitor data transfers and computations, respectively. These in turn use actuators to talk to the various Grid middleware. A metadata manager talks to different information sources such as NWS [50] and the Globus MDS and supplies the gathered data to the scheduler.

APST supports different low-level Grid middleware through the use of actuators and also allows for different scheduling algorithms to be implemented. However, it is focused towards PSAs. APST provides the ability to specify data repositories of different types in the input file and has a separate data manager to manage data transfers.

4.3. Condor-G

Condor-G [11] is a computational management system that allows users to manage multi-domain, heterogeneous resources running Globus [2] and Condor [51] middleware, as though belonging to a single domain. It combines the harnessing of resources in a single administrative domain provided

by Condor with the resource discovery, resource access and security protocols provided by the Globus Toolkit. At the user side, Condor-G provides API and command line tools to submit jobs, cancel them, query their status and to access log files. A new Grid manager daemon is created for each job request, which then submits the job to the remote Globus gatekeeper that starts a new `JobManager` process. Condor-G provides 'execute once' semantics by using a two-phase commit protocol for job submission and completion. Fault tolerance is provided on the submission side by a persistent job queue and on the remote side by keeping persistent state of the active job within the `JobManager`. Jobs are executed on the remote resource within a *mobile sandbox* that traps system calls issued by the task back to the originating system and are check-pointed periodically using Condor mechanisms. This technology, called Condor GlideIn, effectively treats a collection of Grid resources as a Condor pool. Resource brokering is provided by matching user requirements with information available from services such as GRIS and GIIS through the ClassAds [52] mechanism.

Condor-G operates in a Globus and Condor-only environment and installs a virtualization layer at each node at runtime that traps system calls and provides check-pointing facilities. Condor can utilize batch queuing systems such as LSF, PBS and NQE, but only through Globus GRAM protocols. Condor-G provides strong fault tolerance mechanisms as a result of its close integration with the low-level Grid middleware. It implements the two-phase commit protocol for Globus job submission for ensuring that the job is executed only once. Through the GlideIn mechanism, it is able to provide libraries that perform check pointing and job migration and maintains a persistent queue to guard against local failures.

Although Condor-G by itself does not provide any data access functions, it can interface with services such as Kangaroo [53] and Stork [54] that enable it to mediate access to remote files and manage data transfers. Condor-G allows for creation of applications belonging to different models, such as workflows, and supports different scheduling strategies. However, it does not natively support resource costs and has no functions for optimizations based on pricing.

4.4. gLite WMS

gLite [39] is an integrated middleware package for the EGEE project that consists of modules for security, information and management, data and job management services. Here the focus is on the gLite's WMS package that provides access to resources running various middleware such as Globus, Condor and Storage Resource Manager (SRM). gLite treats resources as Compute Elements (CE) or Storage Elements (SEs) depending on whether they are computational or data resources, respectively. Jobs are generally non-interactive and batch oriented. The gLite WMS handles job scheduling and resource allocation and uses Condor-G for job dispatch and management. The WMS accepts job requests and stores them in its task queue. A matchmaker sub-component matches job requests against resource information stored in an Information Super Market (ISM) subcomponent using the Condor ClassAds mechanism. The WMS uses both eager scheduling (jobs are 'pushed' to the resource) and lazy scheduling (resource 'pulls' or requests for jobs). Data required by a job scheduled at a CE are replicated to the nearest SE.

gLite works within a standardized Grid environment running EGEE middleware and has a standardized client configuration that requires external services such as R-GMA (Relational Grid Monitoring Architecture) Information System. gLite is installed on a dedicated machine and accepts job requests from local and remote clients. Thus, it is a centralized resource-brokering system and,

therefore, differs considerably from the other resource brokers which are primarily user-directed, client-focused resource-brokering mechanisms.

gLite automatically schedules replication of the required data for a job with the closest SE to the CE where the job has been scheduled. But, the locations of the data are not taken into account during the selection of the CE itself. That is, gLite does not perform any optimization for reducing the amount of data to be transferred for an execution. gLite interfaces with an accounting module that enables it to keep track of usage and charge users. However, it does not provide any economy-based scheduling of Grid applications.

4.5. Comparison

Table I compares the Gridbus broker and the related work discussed previously against characteristics derived from the challenges listed at the beginning of this chapter. While it may seem unfair to compare the other brokers against requirements that they were not designed for, this comparison is only a discussion of how the design of the Gridbus broker is different and is not a measure of the applicability of the brokers to any situation.

From the table, it can be seen that the design of the Gridbus broker was motivated by different considerations than that of the other brokers. The focus of the Gridbus broker has been on scheduling and executing distributed data-intensive applications on potentially heterogeneous Grid resources. This is in contrast to Condor-G and gLite, which are primarily job management systems, or Nimrod/G, which focuses on computationally intensive PSAs. APST schedules jobs so as to reuse data that have already been transferred but the initial location of the data is the client machine or the machine on which the broker is executing. Also, the Gridbus broker has been designed to enable economy-based strategies for Grid scheduling, going beyond the resource pricing provided by Nimrod-G, by supporting services such as market directories and resource accounting. The Gridbus broker is a single-user system, each application execution requires a different instantiation of the broker. This is different from systems such as gLite, which is a centralized resource broker that handles multiple users.

One of the design principles that differentiate the Gridbus broker from the other resource brokers is the support for different Grid middleware. Except APST, the others work with only Globus services, or, in the case of gLite, with resources running only EGEE middleware. This decoupling in the Gridbus broker has been achieved by limiting all middleware dependencies to the execution layer and by not assuming the presence of specific services or libraries on the remote resources. The benefit of this loose coupling of the broker to low-level Grid middleware is that it can utilize a greater number and range of resources. The object-oriented nature of the broker also makes it easy to support any low-level Grid middleware, if required. For example, for compute Grid middleware, all that is required is to extend the `ComputeServer` and the `JobWrapper` classes. However, this approach has its disadvantages as well. As mentioned before, Condor-G is able to provide stronger fault tolerance semantics due to its close integration with Globus and because it is able to install a virtualization layer that periodically saves the state of jobs on the resources. Such a feature would require the broker to assume the availability of certain libraries on the resources.

Another distinctive design feature of the Gridbus broker is the equality of all types of services, whether they are compute, data or information services. That is, all the Grid services are treated as first-class citizens. This enables the broker to achieve different kinds of strategies, such as those

Table I. Comparison between the broker and related work.

Characteristics	Nimrod/G	APST	Condor-G	gLite	Gridbus
I. Service heterogeneity					
1. Support for different low-level computational middleware	Through Globus, Legion and SSH	Through Globus and SSH	Through Globus and Condor	Only EGEE	Through Globus and SSH as well as Alchemi
2. Support for different Data Grid middleware	None	FTP, GridFTP	Through Stork	Only EGEE	FTP, GridFTP, SRB
3. Equality of different service types	No	No	No	No	Yes
II. Support for application models					
1. Basic application model	Parameter sweep	Parameter sweep	Single job	Single job	Independent tasks (jobs and parameter sweep)
2. Support for workflows	External	External	External	External	Internal
3. Access to internal entities	Through database	Restricted API	Restricted API	Client API	Fully programmable
III. Realization of user objectives					
1. Scheduling based on location of data	No	Yes	No	No	Yes
2. Third-party data transfers	Yes	Yes	Through Stork	Data Replication	Yes
3. Late binding of data locations to jobs	No	Yes	Through Stork and Kangaroo	No	Yes
4. Access to dynamic network information	Yes	Yes	No	No	Yes
5. Resource pricing and cost-based scheduling	Yes	No	No	No	Yes
6. Managing multiple credentials across VOs	No	No	No	No	Yes
IV. Infrastructure					
1. Check pointing of jobs	Yes	No	Yes	Through Condor-G	No
2. Execute once semantics	No	No	Yes	Through Condor-G	Yes
3. Local persistent store	Yes	Yes	Yes	Yes	Yes
4. Dependencies on remote services	Yes	No	No	Yes	No

which give more prominence to data rather than computational requirements. The other brokers, with the exception of APST, focus on the computational aspect of the jobs. While these handle data in different ways—for example, Condor-G presents the data requirements of an application to

Stork to handle while gLite simply replicates it on demand—they do not generally have strategies to choose a specific data repository at runtime based on current network conditions. The Gridbus broker has been designed to provision for such requirements.

5. PERFORMANCE EVALUATION

The Gridbus broker has been evaluated in the context of application deployment in several case studies published previously [12,32,34,55]. In this section, however, we evaluate the performance of the internal elements of the broker alone, both on the client machine and in its interactions with remote middleware. From the previous discussion, it can be seen that the broker is designed to be a general-purpose resource broker for executing large, data-intensive, coarse-grained distributed applications over global Grids. That is, its design was determined by generic requirements of distributed Grid applications and was not tied down to a specific application or remote Grid middleware. The first set of experiments examines the implications of this design on the performance of the broker on the machine on which it is running. However, the broker's performance is determined not only by its architecture and design but also by its interactions with the remote middleware. The second set of experiments evaluates the various overheads incurred due to meta-scheduling and remote execution on the jobs.

In these experiments, we used a 'synthetic' batch job that computes a mathematical function and enables us to vary the job length and data files used for simulating long-running and data-intensive jobs, respectively.

5.1. Performance of the broker on its host

The workload of the broker is determined by its job and the service objects. Therefore, the experiments that follow are divided into two parts: one with increasing number of jobs, and the other with increasing number of compute resources, or `ComputeServer` objects. The metrics measured were the maximum heap memory accessed and the number of threads (transient and persistent) spawned by the broker during the execution. The broker was executed as a command-line program on a machine with Intel Xeon CPU and 2 GB of RAM running Redhat Linux 8.0 with version 1.4.3 of the Java Virtual Machine (JVM) installed. The remote computing resources used in this experiment are given in Table II and contain three clusters that are running job management systems such as PBS and SGE in conjunction with Globus.

Figure 13 shows the graphs of the heap memory accessed by the broker and the average number of threads spawned by the broker during a single execution. In Figure 13(a) the number of jobs was increased from 20 to 2000 and a single compute resource was used throughout, while in Figure 13(b) the number of compute resources was increased from 1 to 4 and the number of jobs was kept constant at 1000.

It can be seen from Figure 13(a) that the average number of threads remains between 25 and 27, while the maximum heap size increases from 29 to 57 MB when the number of jobs increases from 20 to 2000. Therefore, the average number of threads spawned is almost constant and there is only a two times increase in heap memory size for a 100 times increase in workload. In contrast, the average number of threads spawned increases slowly from 25 to 29, while the maximum heap size accessed increases from 30 to 100 MB. This can be explained as below.

Table II. The set of Grid resources used for the experiment.

| Organization	Node details (Hostname, Architecture)	Grid Middleware
University of Melbourne, Australia	`belle.cs.mu.oz.au IBM eServer, 4 CPU, 2GB RAM, 70 GB HD, RH Linux 8.0`	Globus 2.4
University of Melbourne, Australia	`manjra.csse.unimelb.edu.au, Linux Cluster, 13 nodes`	Globus 4.0 + PBS
Victorian Partnership for Advanced Computing, Melbourne	`ng1.vpac.org, Linux Cluster, 16 nodes,`	Globus 2.4 + PBS
Universidad Complutense Madrid, Spain	`aquila.dacya.ucm.es, Linux Cluster`	Globus 4.0 + SGE
Universidad Complutense Madrid, Spain	`draco.dacya.ucm.es`	Globus 4.0
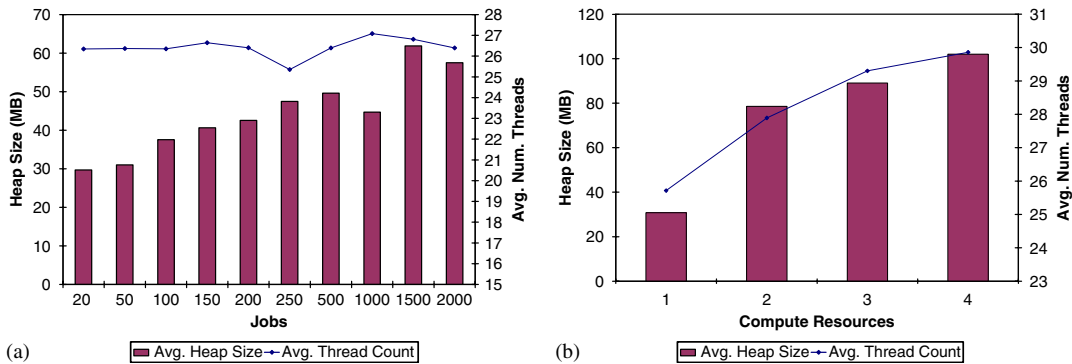
Figure 13. Average heap size and average thread count for the broker for (a) increasing number of jobs and (b) increasing number of compute resources.

The only threads within the broker that are constantly running throughout an execution are the `ServiceMonitor`, the `Scheduler` and the `JobMonitor`. These perform tasks that involve interaction with remote resources, such as probing a compute resource, dispatching a batch of jobs and probing remote jobs by spawning time-limited transient threads. This is so because remote interactions are prone to failures and may cause bottlenecks within the broker if carried out in sequence. Also, the `Scheduler` loads only a subset of jobs (which we term as the 'active set') from the database into the memory at a time. The size of the active set is configurable by the user and also depends on the memory available for the broker. The size of the active set was 20 jobs throughout this experiment. Therefore, the increase in the number of jobs did not translate to a similar increase in the heap size accessed. For the same reason, the number of threads was more or less constant even as the size of the workload (number of jobs) increased. However, when the number of compute resources increases, the number of transient threads probing them increases as well. Also, the corresponding objects (`ComputeServices` and related objects) reside in the

memory throughout an execution. Therefore, the memory usage increases with increasing number of services.

5.2. Interaction with remote middleware

Two experiments were performed, varying over different job profiles and over different middleware. For each variation, the following four metrics were measured:

1. *Time for submission*: This is measured as the time span between the instant the job was 'mapped' to a grid resource by the broker scheduler and the instant when the job handle was retrieved from the remote middleware system. This measure provides the time taken by the dispatcher to submit a mapped job to its chosen compute resource.
2. *Total querying time*: This is the total time spent for a job in querying its status on the remote node. It is measured as the sum of the times taken for each query at periodic intervals.
3. *Time for termination*: This is the time spent for terminating and cleaning up after a job is indicated as completed by the remote middleware. This, therefore, includes the time to retrieve the output files from the resource and the time to delete the directories created by the job on the remote resource.
4. *Job wallclock time*: This is measured as the difference between the time the job was submitted to the remote resource and the time the job was completed.

Each of the metrics is reported as a 'per job' value calculated as an average over a set of 50 jobs. The first experiment used one of the Grid resources from Table II (ng1.vpac.org) running Globus 2.4. The broker was executed as a non-interactive command-line program on a notebook computer (running Microsoft Windows XP, with an Intel Pentium M 2 GHz processor and 2 GB RAM). The application binaries were installed on the remote middleware, and hence the times shown do not include the time required for actually copying the application itself. The tests were run against different job profiles shown in Table III.

Figure 14 shows the distribution of time taken by simple, data-intensive and compute-intensive jobs in various stages of a job execution on a Globus 2.4 resource. The time to dispatch a job to the remote resource is similar for both simple and computationally intensive jobs, but is higher for data-intensive jobs as it includes the time to stage in the input data file to the remote node. The jobs were queried every 12 s and, therefore, the aggregate time spent by the broker to query a job naturally increases with the length of the job. This measure does not affect the wallclock time as the querying occurs on the broker's end. The termination time is almost equal for all the jobs as only the standard output and standard error files for the job are copied in all the three cases.

Table III. Profile of jobs used in experiments.

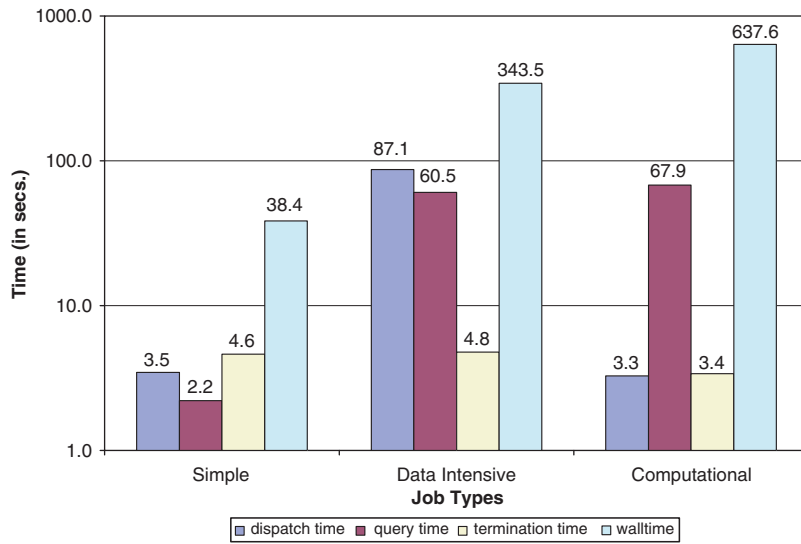| Job profile	Job length (min) (approx.)	Data requirements (I/O) (MB)
Simple	0.5	<1
Data intensive	5	100 (approx.)
Compute intensive	10	<1

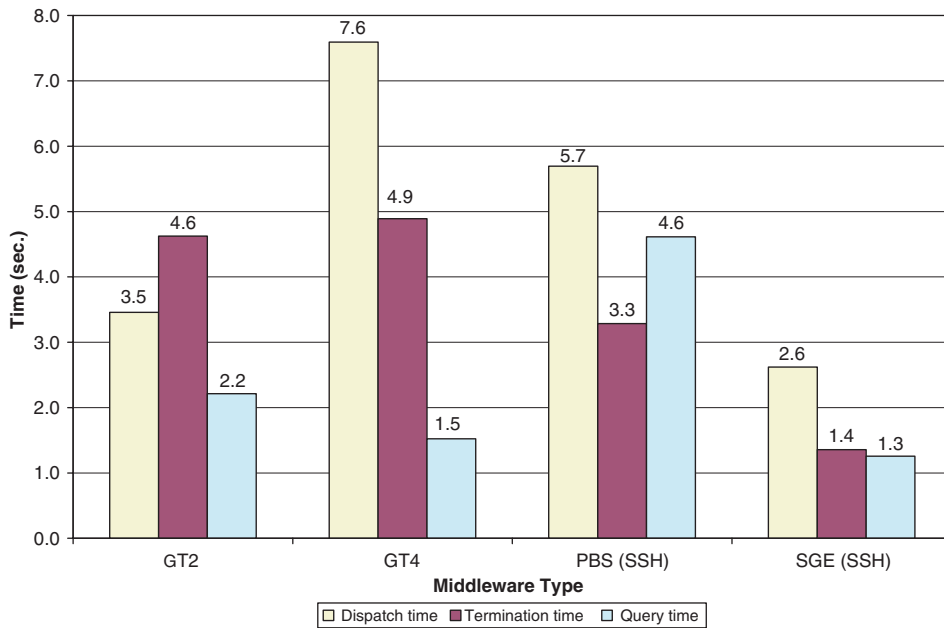Figure 14. A comparison of remote interaction overheads for different job types (GT2).

Figure 15. Comparison of dispatch, termination and query times for different middleware on a single machine.

The last experiment was performed by installing four low-level grid middleware—Globus 2 (GT2), Globus 4 (GT4), PBS and SGE—on a single machine, which in this case was a cluster in our department (`manjra.csse.unimelb.edu.au`). The broker was started on another server machine within our department (`belle.csse.unimelb.edu.au`). All the four middleware were tested with a set of 50 jobs following the simple profile in Table III. PBS and SGE were tested by direct invocation of commands through SSH. Figure 15 displays the results of this experiment.

It can be seen here that the highest dispatch time is observed for GT4, which could be due to the effect of Web-service-based messaging, resulting in extra delays for serializing and deserializing XML messages. GT2 incurs a higher overhead than SGE, which is probably due to the additional burden of security and encryption offered by the GSI libraries. The PBS and SGE dispatchers in the broker were expected to perform on par; however, we see that PBS has incurred higher overheads in all cases. This is an issue specific to the PBS implementation in the broker.

6. LESSONS LEARNT FROM OUR EXPERIENCE

The development of the Gridbus broker with focus on data-intensive applications was initiated in early 2003; since then, it has undergone several version changes and feature additions. While the three-layer architecture has remained unchanged from the very beginning of the broker, the design has been updated several times to include new features and to increase performance. We summarize the lessons that we have learnt in this interval in the following points:

1. *Dealing with heterogeneity*: As mentioned in Section 3.3.1, the broker does not follow a specific interaction model with remote middleware or assume the availability of certain libraries on the resources. The presence of Grid middleware also presented with another layer of heterogeneity on top of the one caused by the presence of multiple operating systems. For example, the Globus Toolkit, which is the most popular Grid middleware in terms of deployment, has progressed from version 2.*x* through to 4.*x* over the span of 5 years. Each of these version changes has involved paradigm shifts—from RPC (Remote Procedure Calls) to OGSI (Open Grid Service Infrastructure) to WSRF—and are, therefore, mostly incompatible with each other. Large Grid infrastructures such as UK e-Science project, TeraGrid in the U.S. and Australia's own APAC Grid have struggled to keep up with these changes and to port Grid applications from one version to another. In many cases, for example, in the EGEE project in Europe and the LHC Grid project, Globus 2.*x* is still used at the time of writing due to the massive investment in applications, personnel and training around the existing infrastructure. The Gridbus broker was initiated at a time when OGSI had just been introduced. Therefore, its architecture and the design evolved out of the need to easily port between different versions of Grid middleware, and were later generalized to cover all different types of middleware.
2. *Thread control*: The first version of the broker created a thread for every active job. However, as we realized soon, this was not scalable and also caused problems with thread synchronization. Also, when used within Web portals, the broker would not exit cleanly sometimes as some of the threads would remain suspended. Eventually, we re-designed the core to have only three main threads which fire off short-lived threads when interacting with remote resources. As mentioned earlier, the transient threads are also provided with time-outs to ensure that they will exit and be removed from the virtual machine (VM).

3. *Handling multiple broker instances*: Up to version 2.2 of the broker, a properties file was provided to set up certain user-configurable parameters (e.g. location of credentials, polling time). Reference to this file (as a Java Properties object) was made static so that any component within the broker was able to access this file during runtime. However, this caused severe problems when two or more brokers were initiated within a Web portal container as they shared the reference to the same Properties object within the same VM. This was remedied by providing persistence and a unique ID for each instance of the broker so that the properties can be referenced by the ID of the instance.

It has to be noted that issues 2 and 3 in the list above arose when the broker was applied to the portal environment. Prior to this, it was only used in command line environments where there was only one instance of the broker running inside a single VM. This is an example of how adapting to different user environments involves solving different challenges.

7.   SUMMARY AND CONCLUSION

In this paper we have presented the architecture and design of Gridbus Broker that is motivated by the challenges and requirements of presenting an abstract interface for users and applications to distributed and heterogeneous Grid resources. We have shown how the presented design meets these requirements and how these differentiate it from the designs of similar Grid resource brokers. We have also presented two experiments that measured the effect of the design choices both on the performance of the broker on its host machine and on its interactions with remote resources. Lastly, we presented the lessons learnt from our development experience.

The Gridbus broker now presents a platform on which enhanced resource-brokering strategies can be developed and deployed. Such strategies may involve use of artificial intelligence techniques for task scheduling, task allocation that takes into account varying prices and resource demand, and resource selection using predictive techniques. One of the requirements for meeting users' QoS guarantees is the ability to reserve resources in advance. At the time of writing, standards are being evolved to enable reservation in resource managers. The broker also needs to work with resources with such abilities. The broker is also being extended to support more application models. For example, the broker was recently modified to provide an environment for describing and executing workflows based on work previously performed in the GRIDS Lab [56]. This was done without modifying the existing entities. This exercise will be the subject of a future publication.

To conclude, it can be stated that the design of the Gridbus broker has been successful in meeting its requirements. It provides an environment which users and developers can exploit to derive the maximum out of the available Grid services without having to worry about dealing with the challenges of heterogeneity and dynamic variations in availability.

REFERENCES

1. Foster I, Kesselman C (eds.). *The Grid*: *Blueprint for a Future Computing Infrastructure*. Morgan Kaufmann: U.S.A., 1999.
2. Foster I, Kesselman C. Globus: A metacomputing infrastructure toolkit. *International Journal of Supercomputer Applications* 1997; **11**(2):115–128.
3. Chapin S, Karpovich J, Grimshaw A. The Legion resource management system. *Proceedings of the 5th Workshop on Job Scheduling Strategies for Parallel Processing* (*JSSPP*), San Juan, Puerto Rico, (*Lecture Notes in Computer Science*, vol. 1659). Springer: Berlin, Germany, 1999.
4. Casanova H, Dongarra J. NetSolve: A network server for solving computational science problems. *Proceedings of the 1996 ACM/IEEE Conference on Supercomputing* (*SC '96*), Pittsburgh, PA, U.S.A. IEEE Computer Society Press: Los Alamitos, CA, U.S.A., 1996.
5. Nakada H, Sato M, Sekiguchi S. Design and implementation of Ninf: Towards a global computing infrastructure. *Future Generation Computing Systems* 1999; **15**(5–6):649–658.
6. Allen G, Benger W, Goodale T, Hege H-C, Lanfermann G, Merzky A, Radke T, Seidel E, Shalf J. The cactus code: A problem solving environment for the grid. *Proceedings of the 9th International Symposium on High Performance Distributed Computing* (*HPDC-9*), Pittsburgh, U.S.A. IEEE Computer Society Press: Los Alamitos, CA, U.S.A., 2000.
7. Berman F, Chien A, Cooper K, Dongarra J, Foster I, Gannon D, Johnsson L, Kennedy K, Kesselman C, Mellor-Crumme J, Reed D, Torczon L, Wolski R. The GrADS project: Software support for high-level grid application development. *International Journal of High Performance Computing Applications* 2001; **15**(4):327–344.
8. Berman F, Wolski R. The AppLeS project: A status report. *Proceedings of the 8th NEC Research Symposium*, Berlin, Germany, 1997.
9. Buyya R, Abramson D, Giddy J. Nimrod/G: An architecture for a resource management and scheduling system in a global computational grid. *Proceedings of the 4th International Conference on High Performance Computing in Asia-Pacific Region* (*HPC Asia 2000*), Beijing, China. IEEE Computer Society Press: Los Alamitos, CA, U.S.A., 2000.
10. Casanova H, Obertelli G, Berman F, Wolski R. The AppLeS parameter sweep template: User-level middleware for the grid. *Proceedings of the 2000 ACM/IEEE Conference on Supercomputing* (*SC'00*), Dallas, TX. IEEE Computer Society Press: Los Alamitos, CA, U.S.A., 2000.
11. Frey J, Tannenbaum T, Livny M, Foster I, Tuecke S. Condor-G: A computation management agent for multi-institutional grids. *Cluster Computing* 2002; **5**(3):237–246.
12. Venugopal S, Buyya R, Winton L. A grid service broker for scheduling e-Science applications on global data grids. *Concurrency and Computation*: *Practice and Experience* 2006; **18**(6):685–699.
13. Foster I, Kesselman C, Tuecke S. The anatomy of the grid: Enabling scalable virtual organizations. *International Journal of High Performance Computing Applications* 2001; **15**(3):200–222.
14. Baker M, Buyya R, Laforenza D. Grids and grid technologies for wide-area distributed computing. *Software—Practice and Experience* 2002; **32**(15):1437–1466.
15. Vraalsen F, Aydt R, Mendes C, Reed D. Performance contracts: Predicting and monitoring grid application behavior. *Proceedings of the 2nd International Workshop on Grid Computing* (*GRID 2001*), Denver, CO, (*Lecture Notes in Computer Science*, vol. 2242). Springer: Berlin, Germany, 2001.
16. Yu J, Venugopal S, Buyya R. A market-oriented grid directory service for publication and discovery of grid service providers and their services. *Journal of Supercomputing* 2006; **36**(1):17–31.
17. Czajkowski K, Fitzgerald S, Foster I, Kesselman C. Grid information services for distributed resource sharing. *Proceedings of 10th IEEE International Symposium on High Performance Distributed Computing* (*HPDC-10*). IEEE Computer Society Press: U.S.A., 2001.
18. Eriksson H-E, Penker M. *Business Modeling with UML*: *Business Patterns at Work*. Wiley: New York, U.S.A., 2000.
19. Bjorkander M, Kobryn C. Architecting systems with UML 2.0. *IEEE Software* 2003; **20**(4):57–61.
20. Abramson D, Sosic R, Giddy J, Hall B. Nimrod: A tool for performing parametrised simulations using distributed workstations. *Proceedings of the 4th IEEE International Symposium on High Performance Distributed Computing* (*HPDC '95*), Pentagon City, VA, U.S.A. IEEE Computer Society Press: Los Alamitos, CA, U.S.A., 1995.
21. Anjomshoaa A, Brisard F, Drescher M, Fellows D, Ly A, McGough S, Pulsipher D, Savva A. Job submission description language (JSDL) specification, version 1.0. *Technical Report*, Global Grid Forum, 2005.

22. von Laszewski G, Foster I, Gawor J, Lane P. A Java commodity grid kit. *Concurrency and Computation*: *Practice and Experience* 2001; **13**(8–9):643–662.
23. Bayucan A, Henderson R, Lesiak C, Mann B, Proett T, Tweten D. Portable batch system: External reference specification. *Technical Report*, MRJ Technology Solutions, 1999.
24. Gentzsch W. Sun grid engine: Towards creating a compute power grid. *Proceedings of the 1st International Symposium on Cluster Computing and the Grid* (*CCGRID '01*), Brisbane, Australia. IEEE Computer Society Press: Los Alamitos, CA, U.S.A., 2001.
25. Luther A, Buyya R, Ranjan R, Venugopal S. Peer-to-peer grid computing and a .NET-based Alchemi framework. *High Performance Computing*: *Paradigm and Infrastructure*. Wiley: U.S.A., 2005.
26. Erwin DW, Snelling DF. UNICORE: A grid computing environment. *Proceedings of the 7th International Euro-Par Conference on Parallel Processing* (*Euro-Par '01*), Manchester, U.K. Springer: Berlin, Germany, 2001.
27. Litzkow M, Livny M, Mutka MW. Condor—A hunter of idle workstations. *Proceedings of the 8th International Conference of Distributed Computing Systems* (*ICDCS 1988*), San Jose, CA, U.S.A. IEEE Computer Society Press: Los Alamitos, CA, U.S.A., 1988.
28. Kramer D, MacInnis M. Utilization of a local grid of Mac OS X-based computers using Xgrid, Poster Paper. *Proceedings of the 13th IEEE International Symposium on High Performance Distributed Computing* (*HPDC '04*), Honolulu, HI, U.S.A. IEEE Computer Society Press: Los Alamitos, CA, U.S.A., 2004.
29. Foster I, Czajkowski K, Ferguson D, Frey J, Graham S, Maguire T, Snelling D, Tuecke S. Modeling and managing state in distributed systems: the role of OGSI and WSRF. *Proceedings of the IEEE* 2005; **93**(3):604–612.
30. Czajkowski K, Foster I, Karonis N, Kesselman C, Martin S, Smith W, Tuecke S. A resource management architecture for metacomputing systems. *Proceedings of the Workshop on Job Scheduling Strategies for Parallel Processing* (*JSSPP*), (*Lecture Notes in Computer Science*, vol. 1459), Feitelson DG, Rudolph L (eds). Springer: Berlin, Germany, 1998.
31. Czajkowski K, Foster I, Kesselman C. Resource co-allocation in computational grids. *Proceedings of the Eighth IEEE International Symposium on High Performance Distributed Computing* (*HPDC '99*), Redondo Beach, CA, U.S.A. IEEE Computer Society Press: Los Alamitos, CA, U.S.A., 1999.
32. Gibbins H, Nadiminti K, Beeson B, Chhabra RK, Smith B, Buyya R. The Australian BioGrid portal: Empowering the molecular docking research community. *Proceedings of the 3rd APAC Conference and Exhibition on Advanced Computing*, *Grid Applications and eResearch* (*APAC 2005*), Gold Coast, Australia, 2005.
33. Dias de Assuncao M, Nadiminti K, Venugopal S, Ma T, Buyya R. An integration of global and enterprise grid computing: Gridbus broker and Xgrid perspective. *Proceedings of the 4th International Conference on Grid and Cooperative Computing* (*GCC 2005*), Beijing, China, (*Lecture Notes in Computer Science*). Springer: Berlin, Germany, 2005.
34. Kolbe S, Ma T, Liu W, Soh W, Buyya R, Egan G. A platform for distributed analysis of neuroimaging data on global grids. *Proceedings of the 1st IEEE International Conference on e-Science and Grid Computing* (*e-Science 2005*), Melbourne, Australia. IEEE Computer Society Press: Los Alamitos, CA, U.S.A., 2005.
35. Soh H, Haque S, Liao W, Nadiminti K, Buyya R. GTPE: A thread programming environment for the grid. *Proceedings of the 13th International Conference on Advanced Computing and Communications* (*ADCOM 2005*), Coimbatore, India, 2005.
36. Buyya R, Abramson D, Giddy J, Stockinger H. Economic models for resource management and scheduling in grid computing. *Concurrency and Computation*: *Practice and Experience* 2002; **14**(13–15):1507–1542.
37. Barmouta A, Buyya R. GridBank: A grid accounting services architecture (GASA) for distributed systems sharing and integration. *Proceedings of Workshop on Internet Computing and E-Commerce*: *17th Annual International Parallel and Distributed Processing Symposium* (*IPDPS 2003*), Nice, France. IEEE Computer Society Press: Los Alamitos, CA, U.S.A., 2003.
38. Venugopal S, Buyya R. A deadline and budget constrained scheduling algorithm for e-Science applications on data grids. *Proceedings of the 6th International Conference on Algorithms and Architectures for Parallel Processing* (*ICA3PP-2005*), Melbourne, Australia, (*Lecture Notes in Computer Science*, vol. 3719). Springer: Berlin, Germany, 2005.
39. Lamanna M, Koblitz B, Chen T, Ueng W, Herrala J, Liko D, Maier A, Moscicki J, Peters A, Orellana F, Pose V, Demichev A, Feichtinger D. Experiences with the gLite Grid Middleware. *Proceedings of Conference on Computing in High Energy Physics* (*CHEP '04*), Interlaken, Switzerland, 2004.
40. Huedo E, Montero RS, Llorente IM. A framework for adaptive execution on grids. *Software—Practice and Experience* 2004; **34**:631–651.
41. Dumitrescu C, Raicu I, Foster I. DI-GRUBER: A distributed approach to grid resource brokering. *Proceedings of the 2005 ACM/IEEE Conference on Supercomputing*, Seattle, WA. IEEE Computer Society Press: Los Alamitos, CA, U.S.A., 2005.
42. In J, Arbree A, Avery P, Cavanaugh R, Katageri S, Ranka S. Sphinx: A scheduling middleware for data intensive applications on a grid. *Technical Report*, *GriPhyN 2003-17*, Grid Physics Network, 2003.
43. Seidel E, Allen G, Merzky A, Nabrzyski J. GridLab—a grid application toolkit and testbed. *Future Generation Computer Systems* 2002; **18**(8):1143–1153.
44. Abramson D, Giddy J, Kotler L. High performance parametric modeling with Nimrod/G: Killer application for the global grid? *Proceedings of the International Parallel and Distributed Processing Symposium* (*IPDPS 2000*), Cancun, Mexico, 1–5 May 2000. IEEE Computer Society Press: U.S.A., 2000.

45. Ayyub S, Abramson D, Enticott C, Garic S, Tan J. Executing large parameter sweep applications on a multi-VO testbed. *Proceedings of the 7th International Symposium on Cluster Computing and Grid* (*CCGrid 2007*), Rio de Janeiro, Brazil, 14–17 May 2007. IEEE Computer Society Press: Los Alamitos, CA, U.S.A.

46. Buyya R, Giddy J, Abramson D. An evaluation of economy-based resource trading and scheduling on computational power grids for parameter sweep applications. *Proceedings of the 2nd Workshop on Active Middleware Services* (*AMS 2000*), Pittsburgh, U.S.A., 2000. IEEE Computer Society Press: Los Alamitos, CA, U.S.A., 2000.

47. Abramson D, Lewis A, Peachy T. Nimrod/O: A tool for automatic design optimization. *Proceedings of the 4th International Conference on Algorithms & Architectures for Parallel Processing* (*ICA3PP 2000*), Hong Kong, 11–13 December 2000.

48. Casanova H, Legrand A, Zagorodnov D, Berman F. Heuristics for scheduling parameter sweep applications in grid environments. *Proceedings of the 9th Heterogeneous Computing Systems Workshop* (*HCW 2000*), Cancun, Mexico. IEEE Computer Society Press: Los Alamitos, CA, U.S.A., 2000.

49. Yang Y, Casanova H. UMR: A multi-round algorithm for scheduling divisible workloads. *Proceedings of the 17th International Symposium on Parallel and Distributed Processing* (*IPDPS '03*), Nice, France, 2003. IEEE Computer Society Press: Los Alamitos, CA, U.S.A., 2003.

50. Wolski R, Spring N, Hayes J. The network weather service: A distributed resource performance forecasting service for metacomputing. *Future Generation Computing Systems* (FGCS) 1999; **15**:757–768.

51. Litzkow M, Livny M, Mutka MW. Condor—A hunter of idle workstations. *Proceedings of the 8th International Conference of Distributed Computing Systems*, San Jose, CA, U.S.A. IEEE Computer Society Press: Los Alamitos, CA, U.S.A., 1988.

52. Raman R, Livny M, Solomon M. Matchmaking: Distributed resource management for high throughput computing. *Proceedings of the 7th IEEE International Symposium on High Performance Distributed Computing* (*HPDC-7 '98*). IEEE Computer Society Press: Los Alamitos, CA, U.S.A., 1998.

53. Thain D, Basney J, Son S, Livny M. The Kangaroo approach to data movement on the grid. *Proceedings of the 10th IEEE Symposium on High Performance Distributed Computing* (*HPDC10*), San Francisco, CA. IEEE Computer Society Press: Los Alamitos, CA, U.S.A., 2001.

54. Kosar T, Livny M. Stork: Making data placement a first class citizen in the grid. *Proceedings of the 24th International Conference on Distributed Computing Systems* (*ICDCS'04*), Tokyo, Japan. IEEE Computer Society Press: Los Alamitos, CA, U.S.A., 2004.

55. Hughes B, Venugopal S, Buyya R. Grid-based indexing of a newswire Corpus. *Proceedings of the 5th IEEE/ACM International Workshop on Grid Computing* (*GRID 2004*), Pittsburgh, 8 November 2004. IEEE Computer Society Press: Los Alamitos, CA, U.S.A., 2004.

56. Yu J, Buyya R. A novel architecture for realizing grid workflow using tuple spaces. *Proceedings of the 5th IEEE/ACM International Workshop on Grid Computing* (*GRID '04*), Pittsburgh, PA, 2004. IEEE Computer Society Press: Los Alamitos, CA, U.S.A., 2004.