The Gridbus Grid Service Broker and Scheduler (2.0) User Guide

Krishna Nadiminti, Srikumar Venugopal, Hussein Gibbins, Rajkumar Buyya
Grid Computing and **D**istributed **S**ystems (GRIDS) Laboratory,
Department of Computer Science and Software Engineering,
The University of Melbourne, Australia
Email:{kna,srikumar,hag,raj}@cs.mu.oz.au

`http://www.gridbus.org/broker`

Contents

1 INTRODUCTION

1.1 Grid Computing

A ”Grid” is a type of parallel and distributed system that enables the sharing, selection, and aggregation of geographically distributed ”autonomous” resources dynamically at runtime depending on their availability, capability, performance, cost, and users’ quality-of-service requirements. It should be noted that Grids aim at exploiting synergies that result from cooperation - ability to share and aggregate distributed computational capabilities and deliver them as service.
The next generation of scientific experiments and studies, popularly called as e-Science, is carried out by large collaborations of researchers distributed around the world engaged in analysis of huge collections of data generated by scientific instruments. Grid computing has emerged as an enabler for e-Science as it permits the creation of virtual organizations that bring together communities with common objectives. Within a community, data collections are stored or replicated on distributed resources to enhance storage capability or efficiency of access. In such an environment, scientists need to have the ability to carry out their studies by transparently accessing distributed data and computational resources. This is where the concept of resource brokers comes into picture.

1.2 Resource Brokers

A Resource on a grid could be any entity that provides access to a service. This could range from Compute servers to databases, scientific instruments, applications and the like. In a heterogeneous environment like a grid, resources are generally owned by different people, communities or organizations with varied administration policies, and capabilities. Naturally obtaining and managing access to these resources is not a simple task. Resource Brokers aim to simplify this process by providing an abstraction layer to users who just want to get their work done. In the field of Grids and distributed systems, resource brokers are software components that let users access heterogeneous resources transparently, without having to worry about availability, access methods, security issues and other policies. The Gridbus resource broker is a resource broker designed with the aim of solving these issues in a simple way.

1.3 Gridbus Broker

The Gridbus broker is designed to support both computational and data grid applications. For example, it has been used to support composition and deployment of neuroscience (compute-intensive) applications and High Energy Physics (Belle) Data Grid applications on Global Grids.The architecture of the broker has emphasis on simplicity, extensibility and platform independence. It is implemented in Java and provides transparent access to grid nodes running various middleware. The main design principles of the broker include:

- Assume Nothing about the environment

  No assumptions are made anywhere in the Broker code as to what to expect from the Grid resource except for one - that the resource provides atleast one way of submitting a job and if running a flavour of Unix will provide atleast a POSIX shell. Also, no assumption is made about resource availability throughout an execution. The implications of this principle have a huge impact throughout the broker such as

  - The broker has no close integration with any of the middleware it supports. It uses the minimum set of services that are required to run a job on a resource supported by the middleware. Advantages of this are:
    * In a Grid with multiple resources configured differently, the broker tries to make use of every resource possible by not imposing a particular configuration requirement. For example, in the case of Globus 2.4, all is required is that the GRAM service be set up properly on the resource.
    * The broker can run jobs on resources with different middleware at the same time.
    * The broker need not be refactored if there is a new version of the middleware.
  - The broker is able to handle gracefully jobs and resources failing throughout an execution. The job wrapper and job monitor code is written to handle every failure status possible. The scheduler does not fail if a resource drops out suddenly.
  - The failure of the broker itself is taken care of by the recovery module if persistence has been configured.

- Client-centric design

  The scheduler has just one target: that is to satisfy the users' requirements especially if the deadline and budget are supplied. Even in the absence of these, the scheduler strives to get the jobs done in the quickest way possible. Thus, resources are evaluated by the scheduler depending on how fast or slow they are executing the jobs submitted by the broker. In keeping with Principle 1, the broker also does not depend on any metrics supplied by the resource - it does its own monitoring.

- Extensibility is the key

  In Grid environments, transient behaviour is not only a feature of the resources but also of the middleware itself. Rapid developments in this still-evolving field have meant that middleware goes through many versions and unfortunately, interface changes are a norm rather than the exception. Also, changing requirements of Grid users require that the broker itself be flexible enough for adding new features or extending old ones. Thus, every possible care has been taken to keep the design modular and clean. The advantages due to this principle:

  - Extending broker to support new middleware is a zip - Requires implementation of only three interfaces. (For more details refer to Programming section)
  - Getting broker to recognize the new information sources is also easy
  - The differences in middleware are invisible to the upper layers such as the scheduler and vice versa. Thus any changes made in one part of the code remain limited to that section and are immediately applicable. For example, after adding a new middleware, the scheduler is immediately able to use any resource using that middleware.
  - XPML is extensible. Adding any new constructs is easy, using the same reflection framework (see Programming Section). You could also do away with XPML altogether and implement your own favorite interface to describe applications.

Fig.1 shows the block diagram of the broker. The main features of the Gridbus Broker (version 2) are:

- Discovery of resources on the grid
- Transparent Access to computational resources running middleware such as Globus 2.4, Globus 3.2(pre-WS), Alchemi (0.8) and Unicore 4.1, queuing systems such as Condor 6.6.9, OpenPBS 2.3. This includes support for all basic services like:
  - Job scheduling and execution for batch jobs
  - Job monitoring and status reporting
  - Gathering output of completed jobs, and directing it to user-defined locations.

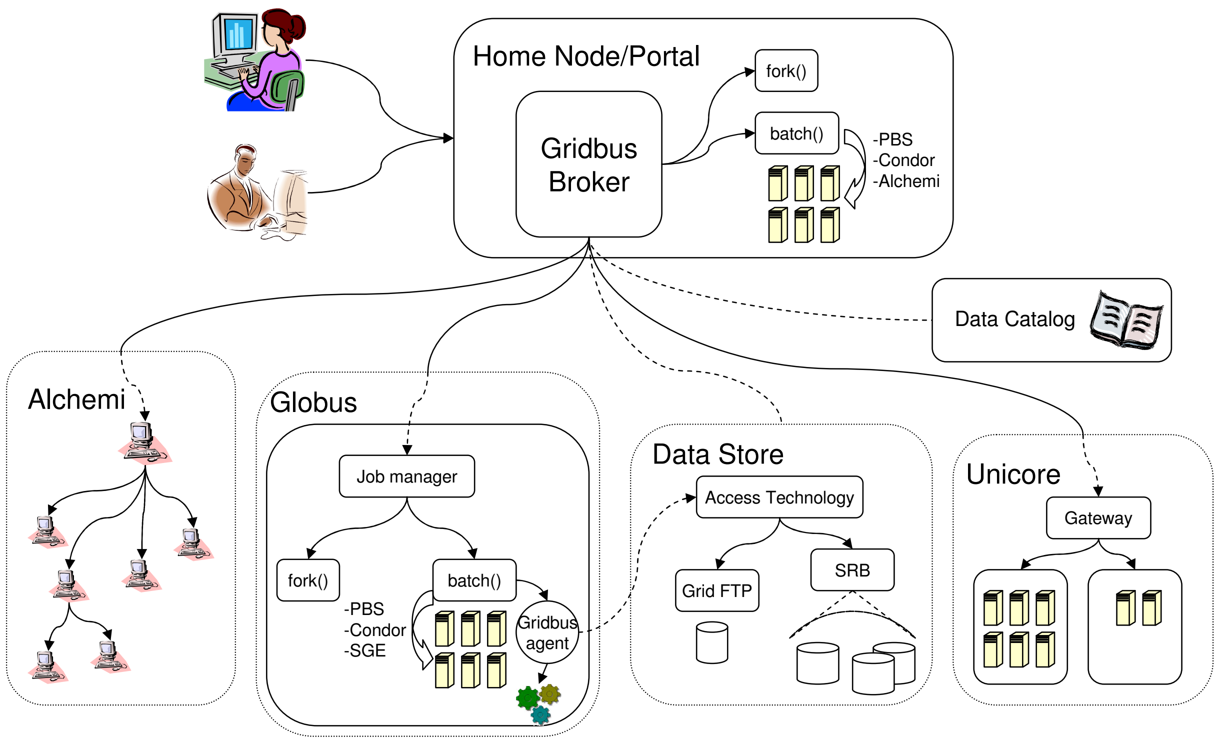
  [Note: Unicore,Condor,PBS support is experimental]
- Economy based scheduling with built-in algorithms for cost, time and cost-time optimizations.
- Data-aware scheduling which considers network bandwidths, and proximity of data to Computational resources
- XML-based application description format
- Support for data sources managed by systems such as Storage Resource Broker (SRB), and the Globus Replica Catalog.
- Support for queuing systems such as PBS on clusters
- Persistence to enable failure management and recovery of an executing grid application
- Extensibility: the broker is engineered to support extensions in the form of custom schedulers, middleware plug-ins, application-description interpreters, and persistence providers.
- Platform independence, which is a natural consequence of a java implementation.

The Gridbus broker comes with a default application-description interpreter for a language called XPML (eXtensible Parametric Modeling Language), which is designed to describe dynamic parameter sweep applications on distributed systems in a declarative way. As such the broker can easily execute parameter-sweep applications on the grid. A parameter sweep application is one in which there is a program which operates on multiple sets of data, and each instance of the running program is independent of the other instances. Such applications are inherently parallel, and can be readily adapted to distributed system. For more information about research on grids and distributed systems please refer to `http://www.gridbus.org`

1.4 Gridbus Broker Architecture

The Gridbus broker follows a service-oriented architecture and is designed on object-oriented principles with a focus on the idea of promoting simplicity, modularity, reusability, extensibility and flexibility. The architecture of the broker is shown in Fig. 2.
The broker can be thought of as a system composed of three main sub-systems:

- the application interface sub-system
- the core-sub-system
- the execution sub-system.



The Gridbus broker works with middleware such as Globus, UNICORE, Alchemi; JobManagers such as Condor, PBS; Data catalogs and also Data storage systems such as the Replica Catalog and SRB .

Figure 1: Broker Block Diagram

The input to the broker is an application-description, which consists of tasks and the associated parameters with their values, and a resource description which could be in the form of a file specifying the hosts available or an information service which the broker queries. At the application interface sub-system there is the application and resource-decription. The app-description interpreter and the resource discovery module convert these inputs into entities, called jobs and servers with which the broker works internally in the core sub-system. A job is an abstraction for a unit of work assigned to a node. It consists of a task, and variables. A variable holds the designated parameter value for a job which are obtained from the process of interpreting the application-description. A server represents a node on the grid, which could provide a compute, storage, information or application service. The task requirements and the resource information drive the discovery of resources such as computational nodes, application and data resources. The resource discovery module connects to the servers to find out if they are available, and if they are suitable for the current application. The broker uses credentials of the user supplied via the resource description, whenever it needs to authenticate with a remote service / server. Once the jobs are prepared and the servers are discovered, the scheduler is started. The scheduler maps jobs (i.e submits jobs using the actuator component in the execution sub-system) to suitable servers based on its algorithm. The actuator is a middleware specific component which dispatches the job to the remote grid node. On receiving a job submission, each server uses its associated server-manager to actuate the middle-ware specific job-submitters (also known as Agents). The job-monitor updates the book-keeper by periodically monitoring the jobs using the services of the execution sub-system.

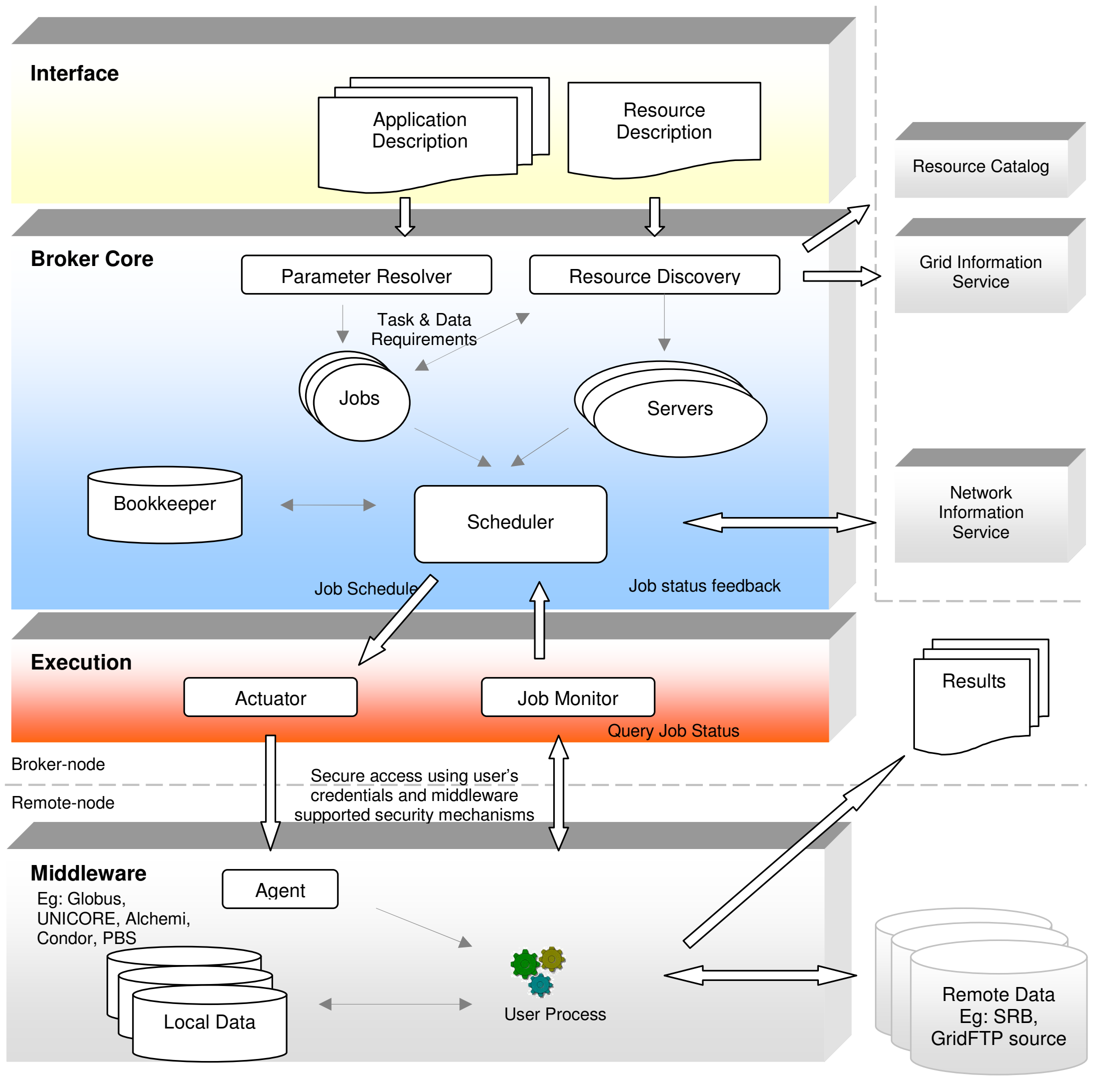


Figure 2: Broker Architecture

As they jobs get completed, the agents take care of clean up and gathering the output of the jobs.The scheduler stops after all the jobs have been scheduled. The scheduling policy determines whether failed jobs are restarted or ignored. Once all scheduling is complete, the broker waits till all jobs are either completed or failed before exiting.

1.5 Sample Applications of the Broker

The Gridbus Broker has been used in Grid-enabling many scientific applications successfully in collaboration with various institutions worldwide. Some of these are listed below:

- Neuroscience(Brain Activity Analysis) [1]- School of Medicine, Osaka University, Japan
- High Energy Physics [2] - School of Physics, University of Melbourne

- Finance (Portfolio analysis) - Complutense University of Madrid, Spain
- Natural Language Engineering [3] - Dept. of Computer Science, University of Melbourne
- Astrophysics [4] - School of Physics, University of Melbourne

It has been also been utilised in serveral Grid demonstrations including the 2003 IEEE/ACM SuperComputing Conference(SC 2003) HPC Challenge demonstration.
The programmer's perspective of the broker design and implementation including the extensibility features, are described in detail in the section 5.

2 INSTALLATION

2.1 Requirements

2.1.1 Broker side (i.e On the machine running the broker)

- Java Virtual Machine 1.4.2 (more info: `http://www.java.com/`)
- Valid certificates properly set up (if using remote Globus nodes)
  By default the certificates are places in the `<USER_HOME>/.globus` directory
  where `<USER_HOME>` is the user's home directory.
  for a user "belle" on a Unix machine this would be:
  `/home/belle/.globus`
  for a user "belle" on a Windows NT/2000/XP machine this would be:
  `C:\Documents and Settings\belle\.globus`
  (For more information on how to acquire and setup x.509 certificates, please consult:
  `http://www.globus.org/security/v1.1/certs.html`)
- Additionally, some ports on the local node should be configured to be open so that the jobs can connect back to the broker. Please refer to the globus documentation for more details.
- Optional Components:
  - Condor v.6.6.9 submit and execute packages (Required if running jobs on a local cluster managed by a Condor system) (more info: `http://www.cs.wisc.edu/condor/downloads/`)
  - OpenPBS v.2.3 (Portable Batch System), (Required if running jobs on a local cluster managed by a PBS system) (more info: `http://www.openpbs.org/`)
  - Network Weather Service (NWS) v.2.8 client tools (Required if running applications that access remote datahosts) (more info: `http://nws.cs.ucsb.edu/`) [Note: NWS client tools are only available for *nix. Grid-applications that need remote data can still be run using the broker on Windows, however, optimal selection of datahosts is not assured, since the absence of NWS will mean the broker cannot get that information by itself. We are working on some way to avoid/workaround this dependency in future versions of the broker.]
  - SCommands Client tools v.3.x (for SRB, Storage Resource Broker) (Required if running applications that need to access SRB data) (more info: `http://www.sdsc.edu/srb/scommands/index.html`

  - access to a MySQL (v.3.x and above) database server installation (either on the machine on which the broker is installed or a remote machine) . (Required if using the persistence feature of the broker. Recommended if you want more robust failure recovery options. The broker will not be able to recover from crashes if persistence is disabled.)

2.1.2 Remote Grid node side

For a compute resource:

- Middleware installation which is one of:
  - Globus 2.4 (more info: `http://www.globus.org`)
  - Globus 3.2 (with the pre-WS globus-gatekeeper and gridftp services running)
  - Alchemi 0.8 (Cross-platform manager) (more info: `http://www.alchemi.net`)
  - Unicore Gateway 4.1 (experimental support within the broker) (more info: `http://www.unicore.org`)
  - Condor 6.6.9 `http://www.cs.wisc.edu/condor/downloads/`
- Optional Components on a compute resource:
  - SRB (SCommands Client tools v.3.x) (Required if running applications that need to access SRB data)

For a data host, one of the following services should be running:

- SRB v.3.x OR
- Globus GridFTP service

Additionally, the user should have permissions to access the remote resources. In case of Globus, the user's credentials should be mapped to an account on the remote node. Please consult the administrator of the resource for more details.

2.2 Installation process

Installing the broker is a simple process. The broker is distributed as a .tar.gz (and a .zip) archive that can be downloaded from `http://www.gridbus.org/broker/2.0/gridbus-broker.2.0.tar.gz` or `http://www.gridbus.org/broker/2.0/gridbus-broker.2.0.zip`. The installation just involves unzipping the files to any directory and optionally setting the `PATH` environment variable to include the broker executable script (gb2.0 or gb2.0.bat depending on your OS). Following are the steps to be followed:

- Unzip the archive to the directory where you want to install the broker. In case of Windows, you can use Winzip (if you download the .zip file) or WinRar (for the .tar.gz) In case of *nix, run the command:
  ```
  $ tar -zxvf gridbusbroker.2.tar.gz
  ```
- The following directory structure is created under the main `gridbus-broker2.0` directory

```
/<broker-install-directory>
    /bin         (contains the broker executable binary)
    /docs        (broker API docs)
    /examples    (example files for using the broker)
    /lib         (all the libraries needed to run the broker)
    /manual      (manual to install and run the broker)
    /src         (the broker source code)
    /xml         (the xml schemas used by the inputs to the broker)
```

- Change `DB_HOME` variable in the `gridbus-broker2.0/bin/classpath.rc` file to point to the directory where you have installed the broker.
- Additionally, it is recommended to have the directory `gridbus-broker2.0/bin` added to the system `PATH` variable. For example, for a Bash shell:
  ```
  $ export PATH=$PATH:<broker-install-directory>/bin
  ```
- Test the installation by running the broker from a shell:
  ```
  $ gb2.0 -test
  ```
  If you see a message confirming that the configuration is ok, congratulations! You have successfully installed the gridbus broker on your machine. This, however, only confirms that you have met the basic requirements on your machine to run the broker. To be sure that all the features of the broker are functional, you will have to make sure the remote resources you have access to, are set up properly. If the test shows any error messages please refer to the ”Troubleshooting” section of this manual.

3 GETTING STARTED USING THE BROKER

The Broker can be used as a stand-alone command-line program or it can be used in your own Java programs or portals. This section describes the use of the Gridbus Broker in both modes.

3.1 Command Line Interface (CLI - running the broker as a stand-alone program)

The broker can be invoked from the command line just like any other java program. The broker distribution comes with a shell script (and a batch file for Windows) which just sets the correct classpath and then calls the broker with any command-line args passed to it. In this mode the broker outputs its messages to both the console and a log file by default. This behaviour can be modified by change the settings in the `Broker.properties` configuration file.

When running the broker on the command line, it needs the following inputs:

- **The Application Description**: The Application description is provided to the broker as an XPML file which describes the grid application.The value for this input can be any absolute or relative path. The broker distribution comes with some sample app-description files found in the examples directory. For example: `examples/calc/calc.xml`
- **The Resource Description**: The Resource description specifies the available resources and describes their attributes. The broker is pointed to the location of the resource list file which contains the resource description in a xml format. The resource description file has a description of the resources that are to be used by the broker for executing the grid application. The broker distribution has a sample

set of resources which are used by us for testing. In almost all cases, this file may have to be modified to specify the resources the user has access to. For example: `examples/calc/resources.xml`

The following instructions assume the broker is being started from the directory where it was installed since it uses relative paths to refer to the broker input files. It also assumes that the `PATH` variable includes the broker binary. To run the broker with the default configuration, the following command is used at the command prompt from the broker's installation directory:

for *nix:

```
<broker-install-dir>$ gb2.0 -a=examples/calc/calc.xml -r=examples/calc/resources.xml
```

for Windows:

```
C:\<broker-install-dir> gb2.0.bat -a=examples\calc\calc.xml -r=examples\calc\resources.xml
```

where `<broker-install-dir>` refers to the directory where the broker is installed.

This will now start the broker, and there should be some output scrolling by, which informs the user about what the broker is doing. For more detailed description about available command-line options/flags, please refer to the ”User Manual” section. If invoked via the command-line, the broker is always a non-interactive program. This behaviour can be altered to suit the user's needs by using the broker APIs in programs built on top of the broker. The next section has some information about how to do that.

3.2 Application Programming Interface(API)

The Gridbus broker is designed to be very flexible and extensible. It is targeted at both basic usage and customisation, by providing programmers the ability to access most of the common APIs which are used internally. Starting from the current version (v.2.0) the full functionality of the XPML files are available for programmatic access via the API. This makes it easy to integrate the broker into your own programs. Using the broker in your programs is as simple as copying the gridbroker.jar into a place where the libraries for your program are located and invoking a single class to start the broker (in the simplest case, as shown below).

```
try{

  //Create a new "Farming Engine"
  GridbusFarmingEngine fe=new GridbusFarmingEngine();

  //Set the Application-description file
  fe.setAppDescriptionFile("calc.xml");

  //Set the Resource-description file
  fe.setResourceDescriptionFile("resources.xml");

  //Call the initialise method to setup jobs and servers
  fe.init();

  //Start scheduling
  fe.schedule();
```

```
    /*
     * The schedule method returns immediately after starting the
     * scheduling. To wait for results / monitor jobs,
     * use the following loop:
     */
    while (!fe.isSchedulingComplete());

  }catch (Exception e){
    e.printStackTrace();
  }
```

The samples provided with the broker distribution show some common ways in which the broker can be invoked from a java program or a jsp application. The Programmers Manual section has a more detailed explanation of how to use the common broker APIs. Programmers who want to use the APIs are suggested to first read through the ”User Manual” section and then go on to the ”Programmers Manual” section which has more detailed explanation of the broker architecture and common APIs. The last section of the ”Programmers Manual” also has descriptions of how to extend the broker’s APIs and build upon them to suit your needs.

4 END-USER GUIDE

4.1 Using the broker on the CLI with various flags

The broker provides the following usage options on the command-line:

```
 gridbus-broker [-mode=startUpMode[-brokerID=<ID>]][-appdesc=XPML file
name][-bc=BrokerProperties file name][-resources=resources.xml or
resource list file name]

-help, -h     : Displays help on using the Broker on the
                command-line. This option cannot be used in combination with any other option.
-test, -t     : Tests the Broker installation and configuration. This
                option cannot be used in combination with any other option.
-version, -v  : Displays the Broker version number. This option
                cannot be used in combination with any other option.
-mode, -m     : Sets the start up mode of the Broker. This option can
                take the following values: "cli", "recover". If ommitted, it
                defaults to "cli" (command-line). If the mode is set to "recover",
                then -brokerID option is mandatory. The Broker looks to recover that
                broker instance whose brokerID is given in the -brokerID option.
                Currently the broker supports RDBMS-based persistence. So, the
                database connection information needs to be specified in the Broker
                configuration file i.e Broker.properties, for recovery mode to work.
-appdesc, -a  : Specifies the app-description file to use. This
                option overrides the APP_DESC_FILE setting in the Broker.properties
                config file.
-resources, -r : Specifies the resource description file. This option
                overrides the RESOURCE_DESC_FILE setting in the Broker.properties
                config file.
```

```
-brokerconfig, -bc : Specifies the Broker configuration file to use.
                     If ommitted, the broker looks for a config file named
                    Broker.properties in the current directory.
```

4.2 The Broker input and configuration files

The main input input and config files that are needed by the broker are as follows:

- The Broker.properties configuration file
- The XPML application description file format
- The Resource description file format

Each of these files and their purpose is described in the subsections below.

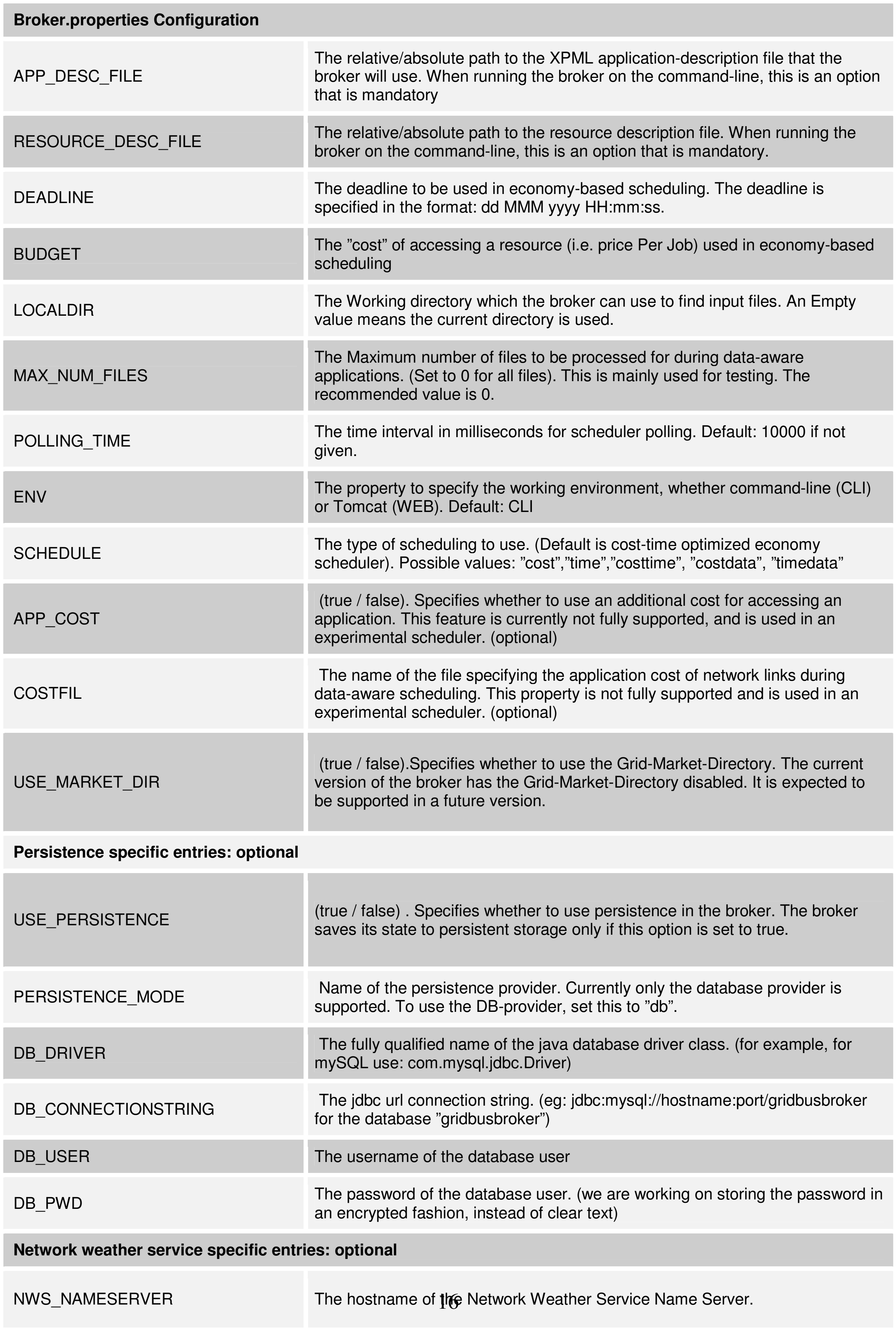
4.2.1 The Broker.properties configuration file

The Broker is configured via a standard java properties file (which is just a plain text file with a name=value pairs one on each line). The default Broker.properties file supplied with the distribution is shown below:

```
# Name of application description file
APP_DESC_FILE=examples/calc/calc.xml

# Name of resource description file
RESOURCE_DESC_FILE=examples/calc/resources.xml

DEADLINE=30 Dec 2010 22:16:00
BUDGET=500000.00

# Working directory. Empty value uses the current directory.
LOCALDIR=

# Maximum number of files to be processed for analysis: set to 0 for all files..
MAX_NUM_FILES=3

#The time interval in milliseconds for scheduler polling
POLLING_TIME=10000

#property to specify the working environment, whether command-line (cli) or tomcat (web)
ENV=cli

SCHEDULE=default
APP_COST=false
COSTFIL=costfil.txt

#Persistence specific entries: optional
USE_PERSISTENCE=true
PERSISTENCE_MODE=db
DB_DRIVER=com.mysql.jdbc.Driver
DB_CONNECTIONSTRING=jdbc:mysql://localhost:3306/gridbusbroker
DB_USER=brokeruser
DB_PWD=somepasswordhere
```

```
NWS_NAMESERVER=belle.cs.mu.oz.au

#-------------------------------------------------
#              Log4j Properties
#-------------------------------------------------

# Root Logger Appenders, Levels : commented out since cog outputs
# everything too... which we dont want
log4j.rootLogger=FATAL, RL
log4j.logger.org.gridbus.broker=INFO, stdout, DB
log4j.logger.org.gridbus.broker.test=DEBUG,stdout
log4j.logger.org.gridbus.broker.persistence=INFO
log4j.logger.org.gridbus.broker.xpml=INFO

log4j.appender.stdout=org.apache.log4j.ConsoleAppender
log4j.appender.stdout.layout=org.apache.log4j.PatternLayout

# Pattern to output the caller's file name and line number.
log4j.appender.stdout.layout.ConversionPattern=%5p (%F:%L) - %m%n

log4j.appender.DB=org.apache.log4j.RollingFileAppender
log4j.appender.DB.File=brokerLogs/broker.log

log4j.appender.RL=org.apache.log4j.RollingFileAppender
log4j.appender.RL.File=brokerLogs/rootlogger.log

log4j.appender.DB.MaxFileSize=1000KB
log4j.appender.RL.MaxFileSize=1000KB

# Keep one backup file
log4j.appender.DB.MaxBackupIndex=1

log4j.appender.DB.layout=org.apache.log4j.PatternLayout
log4j.appender.DB.layout.ConversionPattern=%d %p [%t] (%F:%L) - %m%n

log4j.appender.RL.layout=org.apache.log4j.PatternLayout
log4j.appender.RL.layout.ConversionPattern=%p %t %c - %m%n - (%F:%L)
```

In the Broker.properties config file show above, the broker is configured to look for the application-description file named `calc.xml` in the `examples/calc` directory relative to the current directory from where the broker is executing. The resource description option points to the `resources.xml` in the same directory in this case.
[Note: The file names are case-sensitive or not depending on each operating system. *nix-es are case-sensitive. Win9x is not. Win NT/2000/XP preserves case, but ignores them when reading / accessing the files. It is advised, in general, to pay attention to the "case" of the name of these files always, and set the values accordingly.]
The `Broker.properties` config file has two sections: One dealing with the Broker configuration. The other deals with the lo4j logger configuration which is used by the user for logging output. The Broker configuration file ignores the lines starting with a "#", and considers them as comments. Please note that the options are all specified in upper-case and the broker is particular about the case-sensitivity.
The configuration options available are described in the tables shown in Fig. 3.



Broker.properties Configuration	
APP_DESC_FILE	The relative/absolute path to the XPML application-description file that the broker will use. When running the broker on the command-line, this is an option that is mandatory
RESOURCE_DESC_FILE	The relative/absolute path to the resource description file. When running the broker on the command-line, this is an option that is mandatory.
DEADLINE	The deadline to be used in economy-based scheduling. The deadline is specified in the format: dd MMM yyyy HH:mm:ss.
BUDGET	The ”cost” of accessing a resource (i.e. price Per Job) used in economy-based scheduling
LOCALDIR	The Working directory which the broker can use to find input files. An Empty value means the current directory is used.
MAX_NUM_FILES	The Maximum number of files to be processed for during data-aware applications. (Set to 0 for all files). This is mainly used for testing. The recommended value is 0.
POLLING_TIME	The time interval in milliseconds for scheduler polling. Default: 10000 if not given.
ENV	The property to specify the working environment, whether command-line (CLI) or Tomcat (WEB). Default: CLI
SCHEDULE	The type of scheduling to use. (Default is cost-time optimized economy scheduler). Possible values: ”cost”,”time”,”costtime”, ”costdata”, ”timedata”
APP_COST	(true / false). Specifies whether to use an additional cost for accessing an application. This feature is currently not fully supported, and is used in an experimental scheduler. (optional)
COSTFIL	The name of the file specifying the application cost of network links during data-aware scheduling. This property is not fully supported and is used in an experimental scheduler. (optional)
USE_MARKET_DIR	(true / false).Specifies whether to use the Grid-Market-Directory. The current version of the broker has the Grid-Market-Directory disabled. It is expected to be supported in a future version.
Persistence specific entries: optional	
USE_PERSISTENCE	(true / false) . Specifies whether to use persistence in the broker. The broker saves its state to persistent storage only if this option is set to true.
PERSISTENCE_MODE	Name of the persistence provider. Currently only the database provider is supported. To use the DB-provider, set this to ”db”.
DB_DRIVER	The fully qualified name of the java database driver class. (for example, for mySQL use: com.mysql.jdbc.Driver)
DB_CONNECTIONSTRING	The jdbc url connection string. (eg: jdbc:mysql://hostname:port/gridbusbroker for the database ”gridbusbroker”)
DB_USER	The username of the database user
DB_PWD	The password of the database user. (we are working on storing the password in an encrypted fashion, instead of clear text)
Network weather service specific entries: optional	
NWS_NAMESERVER	The hostname of the Network Weather Service Name Server.

Figure 3: Broker.properties Configuration.

4.2.2 The XPML application description file format

XPML (eXtensible Parametric Modelling Language) is an XML-based language, which is used by the broker to create jobs. Simply put, an XPML application description file is an XML file with special elements as defined in the XMLSchema that comes with the broker. XPML supports description of parameter sweep application execution model in which the same application is run for different values of input parameters often expressed as ranges. A simple application description file is shown below:

```
<?xml version="1.0" encoding="UTF-8"?>
<xpml xmlns:xsi="http://www.w3.org/2001/XMLSchema-instance"
    xsi:noNamespaceSchemaLocation="XMLInputSchema.xsd">
  <parameter name="X" type="integer" domain="range">
    <range from="1" to="3" type="step" interval="1"/>
  </parameter>
  <parameter name="time_base_value" type="integer" domain="single">
    <single value="0"/>
  </parameter>
  <task type="main">
    <copy>
      <source location="local" file="calc"/>
        <destination location="node" file="calc"/>
    </copy>
    <execute location="node">
       <command value="./calc"/>
       <arg value="$X"/>
       <arg value="$time_base_value"/>
    </execute>
    <copy>
       <source location="node" file="output"/>
       <destination location="local" file="output.$jobname"/>
    </copy>
  </task>
</xpml>
```

An XPML app-description consists of three sections: ”parameters”, ”tasks”, and ”requirements”. Parameters: Each parameter has a name, type and domain and any additional attributes. Parameters can be of various types including: integer,string,gridfile and belong to a ”domain” such as: single, range, or file.

- A ”single” domain parameter specifies a variable with just one value which is of the type specified in the ”type” attribute of the parameter element.
- A ”range” domain specifies a parameter which can take a range of values. A range domain parameter has a range element inside it. The range child element has ”from”, and ”to” and ”step” attributes, which specify the starting, ending and step values of the range.
- A ”file” domain parameter specifies a gridfile which is the url of a remote grid file. A gridfile url can have embedded wildcards which are resolved to the actual physical file names by the broker file-resolver. A gridfile url currently supports the url protocols: lfn / srb.

[Note: The name of the child element must match with the value of the domain attribute of the parameter element.] A grid application can any number of parameters. The number of jobs created is the product of the number of all possible values for each parameter. In the example show above, parameter X ranges from 1 to 3. The second parameter has a constant value ”0”. So, the number of jobs created is 3 x 1 = 3 jobs. (since the first parameter can take 3 possible values, and the second parameter can have one possible value). In case ”gridfile” type parameters are used, the number of jobs can be ascertained only at runtime, since the broker has to actually resolve the file names to physical files before creating one job for each file. A ”gridfile” parameter can be defined as shown below.

```
<parameter name="infile" type="gridfile"
  domain="file">
    <file protocol="srb" mode="block" url="srb:/db*.jar" >
</parameter>
```

For multiple gridfiles, multiple <file >elements are placed within the <parameter >element, as shown:

```
<parameter name="infile" type="gridfile" domain="file">
  <file protocol="srb" mode="block" url="srb:/db*.jar"/>
  <file protocol="lfn" mode="block" url="lfn:/somedirectory/someotherdirectory/abc*.txt"/>
  <file protocol="srb" mode="block" url="srb:/sample/example/gridfile/stdout.j*"/>
</parameter>
```

An application can have only one task, with any number of commands in any order. Tasks: A task consists of ”commands” such as copy,gcopy,mcopy,execute,substitute etc.

- A ”copy”/”gcopy”/”mcopy” command specifies a copy operation to be performed. Each of the copy commands has a source and destination file specified. The gcopy supports copying files to and from remote locations. The mcopy command can copy complete directories and supports use of the wildcard character (*).

- An ”execute” command is where actual execution happens. The execute command specifies an executable to be run on the remote node. It also specifies any arguments to be passed to the command on the command-line.

- A ”substitute” command specifies a string substitution inside a text file. This operation is used to substitute the names of user-defined variables, for example. Parameter names can be used as variables with a ”$” pre-fixed. Apart from this, certain special default variables are also defined, such as :$OS (which specifies the Operating system on the remote node), $jobname, which refers to the job ID of a job created by the broker.

The example XPML file shown above specifies a task with three commands. For the grid application described in the file above there are no ”requirements”. With this application-description, the broker creates 3 jobs, with job IDs j1, j2, j3. Each job performs the same set of operations (or commands) as specified in the ”tasks” section. A copy command has a source and a destination child element each with attributes: location, and file. The location can take the values: ”local” and ”node”. The ”local” value is interpreted as the machine on which the broker is executing, and ”node” is understood by the broker as the remote node on which the job is going to execute. These values are substituted at runtime. The ”mcopy” and ”gcopy” also have the same attributes.

For the mcopy command, the source location must be ”node”, the source file name can include wildcards, and the destination must be a directory. For the gcopy command, the source and destination can be both remote, so that third party transfers are possible without the broker intervening in between. The gcopy currently supports the gsiftp protocol and is tailored to work with Globus only. (However, in future versions of the broker, all the protocols ftp,http,https,gsiftp,srb will be eventually supported.)
A substitute command is meant for substitution of parameter (also known as ”variables”) values, in local files which are then used during local operations / remote job execution. Typically, the values of the parameters are determined at runtime, and there could be scenarios in which certain input text files need to be tailored for each job using these parameter values. Any of the parameters can be used as a variable in the files used in a substitute command by pre-fixing ”$” to the parameter name. So, the parameter X, is the variable $ X. A substitute command has source and destination file names, and a location attribute which must be ”local”. The following is an example of a substitute command:

```
    <substitute location="local">
        <source file="input">
        <destination file="input.$jobname>
    </substitute>
```

In the substitute command shown above, the destination element itself has another variable ”$jobname” which refers to the job’s unique id. So, after substitution, the input file is tailored to each job and saved as input.j1, input.j2 etc... for each job. Requirements: Certain jobs need a particular environment during execution. This environment needs to be setup before the job actually starts executing on the remote node. For this purpose, the ”Requirements” element is provided. It can be used to specify a set of initialisation tasks, (and, in the future, conditions). Requirements are of two types: node, and job.
A ”node” requirement is a set of tasks/conditions that need to be satisfied before a node can be used for submission of jobs. So, a node-requirement is performed by the broker before any jobs are submitted to it. This is done once and only once for each node. A ”job” requirement is also a set of tasks/conditions which are to be performed once for each job. Job requirements are currently not implemented in v.2.0 of the broker. Requirements can be specified as follows:

```
    <requirement type="node">
        <!-- anything that can go inside a <task> element can go here -->
    </requirement>
```

The type can be ”node” or ”job”. As mentioned, only ”node” is currently supported for this version. The requirements element can have any number of commands in any order. It is similar to the ”task” element in that respect. The XPML language is undergoing improvements and refinements. It is planned to include more advanced features like <if >conditions, loops, enumerated <range >lists, etc... in future versions.
For those interested, a detailed description of the XPML language schema and the interpretation process is given in the Programmer’s manual section.

4.2.3 The Resource description file

The Resource description file is just an xml file describing the resources that can be used by the broker, and their properties as defined in the resource description schema that comes with the broker. The Resource description can be used to describe two types of entities - resources and credentials (to access the resources). A resource, as defined currently can be of three types:

- compute resources
- storage resource (which function as data sinks)
- services

A sample resource description is shown below:

```
<?xml version="1.0" encoding="UTF-8"?>
<xgrl
xmlns:xsi="http://www.w3.org/2001/XMLSchema-instance"
xsi:noNamespaceSchemaLocation="ResourceDescriptionSchema.xsd">

    <credentials id="prox1" type="proxyCertificate">
        <proxyCertificate source="local">
            <local password="proxyPwd"></local>
        </proxyCertificate>
    </credentials>

    <credentials id="prox2" type="proxyCertificate">
        <proxyCertificate source="myProxy">
            <myProxy password="test" username="test"
                host="myownserver.org"/>
        </proxyCertificate>
    </credentials>

    <credentials id="auth1" type="auth">
        <auth username="test" password="pwd"/>
    </credentials>

    <credentials id="ks1" type="keystore">
        <keystore file="somefile" password="pwd"/>
    </credentials>

    <resource type="compute" credential="auth1">
        <compute domain="local">
            <local jobmanager="pbs"></local>
        </compute>
    </resource>

    <resource type="compute" credential="prox1">
        <compute domain="remote">
            <remote middleware="globus">
                <globus hostname="belle.cs.mu.oz.au"></globus>
            </remote>
        </compute>
    </resource>

    <resource type="compute" credential="ks1">
        <compute domain="remote">
            <remote middleware="unicore">
                <unicore gatewayURL="ssl://hostname.cs.mu.oz.au/"/>
            </remote>
        </compute>
    </resource>

    <resource type="service" credential="prox2">
        <service type="information">
            <information type="srbMCAT">
                <srbMCAT host="srbhost.cs.mu.oz.au" defaultResource="defres"
                    domain="dom" home="myhom" port="9999"  />
            </information>
        </service>
    </resource>

    <resource type="service" credential="auth1">
        <service type="information">
            <information type="replicaCatalog">
                <replicaCatalog replicaTop="top"
                    replicaHost="hostname.gridbus.org"/>
            </information>
        </service>
    </resource>

</xgrl>
```

[Note: One can observe here, that the value of an attribute of a parent element generally determines which child element can be place inside it. For example, the ”type” attribute of a ”resource” element, determines whether a ”compute”, ”storage” or ”service” child element can appear within a compute element. Likewise, the ”domain” attribute of a ”compute” element determines which of ”local”,”remote” becomes its child, and so on. This pattern is followed throughout the resource description schema. Also, the ”credential” attribute of any resource element is of particular significance. Its value points to the id of the credential which is to be used for that resource. The seperation of credential and resource elements within the schema helps to specify a description where the same credential (such as a proxy certificate or a username/password pair) is to be used for authenticating to more than one resource.(It is common experience that such a situation is frequently seen.)]
Compute resources are servers to which the users’ jobs can be submitted for execution. Storage resources are used to store the results of execution, and hence can be considered as data sinks. Service resources are those which provide generic services that can be used by the broker.
A ”compute” resource is associated with a ”domain” which can take two values - ”local” and ”remote”. Local resources could be the local computer, or a cluster (on which the broker is running). It could be used to describe a resource which is running job-management systems such as Condor, PBS, Alchemi. The user can also optionally specify a list of queues which need to be used on the local jobmanager. Remote compute resources are used to represent nodes on the grid which have a job-submission interface accessible via a network. So resources which run grid-middleware such as Globus, Unicore, Alchemi etc. are described here.
A local compute resource can described as follows:

```
    <resource type="compute" credential="auth1">
        <compute domain="local">
            <local jobmanager="pbs" />
        </compute>
    </resource>
```

The jobmanager attribute can be any of PBS,Fork,Alchemi,Condor. In case of PBS,Condor on the local compute resource, the broker would use the configuration of the Condor/PBS client. For the Alchemi manager on a local node, one would need tp specify the credentials of the user. No credentials would be needed to fork jobs on the local node, however the attribute still needs to be provided for the sake of complete conformance with the schema.
To describe a remote, globus node a description similar to the following, is used:

```
<resource type="compute" credential="prox2">
    <compute domain="remote">
        <remote middleware="globus">
            <globus hostname="manjra.cs.mu.oz.au" version="3.2" />
        </remote>
    </compute>
</resource>
```

In the above resource description, the node manjra.cs.mu.oz.au is specified to be running globus v.3.2, and uses the credential identified by the string ”prox2”. This would be the id of a ”proxy”-type credential defined elsewhere in the resource description file. Similarly other compute resources can be described as defined in the schema.
A ”storage” resource is a data sink where the user can opt to store the results of execution of a grid application. Currently this feature is not fully supported by the broker. Hence, a full description of this element is not given here. The user is referred to the schema for more information. A future broker version will have an expanded description, and implementation of storage resources. [Note: Data sources are discovered by the broker at runtime, using the application description which contains file parameters, and information catalog services defined as ”service” elements in the resource description. Hence, the need for explicitly specifying data sources in the resource description is obviated.]
A ”service” resource can be of two types - ”information” services and ”application” services. Information services are typically entities which provide information about other resources or services. These could be LDAP directories, web services, data catalogs etc. Currently supported service types include the SRB MCAT and the Replica Catalog. Application services provide applications hosted on nodes that can be accessed as a service.
The example below shows the definition of a SRB Metadata Catalog. This is modelled as an information service which can be queried to extract data about available SRB storage resources and files located on them.

```
<resource type="service" credential="prox2">
    <service type="information">
        <information type="srbMCAT">
            <srbMCAT host="srbhost.cs.mu.oz.au" defaultResource="defres"
            domain="dom" home="myhom" port="9999"  />
        </information>
    </service>
</resource>
```

A ”credentials” entry describes the user’s authentication information that is used to access the services provided by a grid resource. Credentials can be of the following types - x.509 based proxy certificates, simple username/password pairs, MyProxy saved proxies or keystores.
A proxy credential can be described as follows:

```
<credentials id="prox1" type="proxyCertificate">
    <proxyCertificate source="local">
```

```
            <local password="proxyPwd"></local>
        </proxyCertificate>
    </credentials>
```

Optionally, the location of the usercert and userkey files can also be specified as attributes of the "local" element in the "proxyCertificate".
A simple username/password -type credential can be described as follows:

```
    <credentials id="auth1" type="auth">
        <auth username="test" password="pwd"/>
    </credentials>
```

Every credential has to be given a unique "id" which is referred to by one or more resource elements, as mentioned earlier.

4.3 How to set up persistence

The broker comes with a default persistence-provider, which saves the full broker-state periodically to a mySQL database. This DB(database) persistence provider is implemented using JDBC, and hence can support any JDBC compliant database in general. This section describes the procedure for configuring the broker to use persistence. The procedure to enable persistence involves two steps:

- Installing and configuring the database
- Setting the appropriate properties in the configuration file (`Broker.properties`)

First, the user needs access to a local or remote database, which has the broker database installed and configured for proper access. To install the gridbus broker database, the `broker.sql` script, included in the distribution can be used. This will create a new database named "gridbusbroker" on the database server. Additionally, an initialisation script named init.sql, provided with the broker distribution needs to be run after the creation of the database. This sets up some initial values in the database, and prepares it for use by the broker. It is recommended to create a seperate user login on the database server, for enabling access to this database. The second step is to set the following properties in the broker config file, for example:

```
    USE_PERSISTENCE=true
    PERSISTENCE_MODE=db
    DB_DRIVER=com.mysql.jdbc.Driver
    DB_CONNECTIONSTRING=jdbc:mysql://somehost.cs.mu.oz.au:3306/gridbusbroker
    DB_USER=brokeruser
    DB_PWD=gridbus
```

In the above example, the broker is configured to use the "db" or database persistence provider. (Currently this is the only provider the broker supports.) The `DB_DRIVER` specifies the JDBC driver class to used to connect to the database. The `DB_CONNECTIONSTRING` specifies the string used to connect to the database server. The `DB_USER`, `DB_PWD` set the username and password for the database login.
To disable persistence, just set the `USE_PERSISTENCE` property to "false".

5 PROGRAMMER'S GUIDE

5.1 Design and Implementation

The Gridbus broker has been implemented in Java so that it can be used from the command line, deployed in Web-enabled environments such as Tomcat-driven portals and portlets, and also be used as an API within other programs built on top of the services provided by the broker. It interfaces to nodes running Globus using the Java Commodity Grid (CoG) Kit, to Alchemi nodes using the Alchemi Cross-Platform Manager Interface (or the Alchemi command-line client, if the Manager is not running as a web service), to UNICORE using the arcon client library, to Condor and PBS using their respective command-line clients. This section describes the main APIs and concepts related to the common broker APIs. Fig. 4 below shows some of the main classes in the broker.

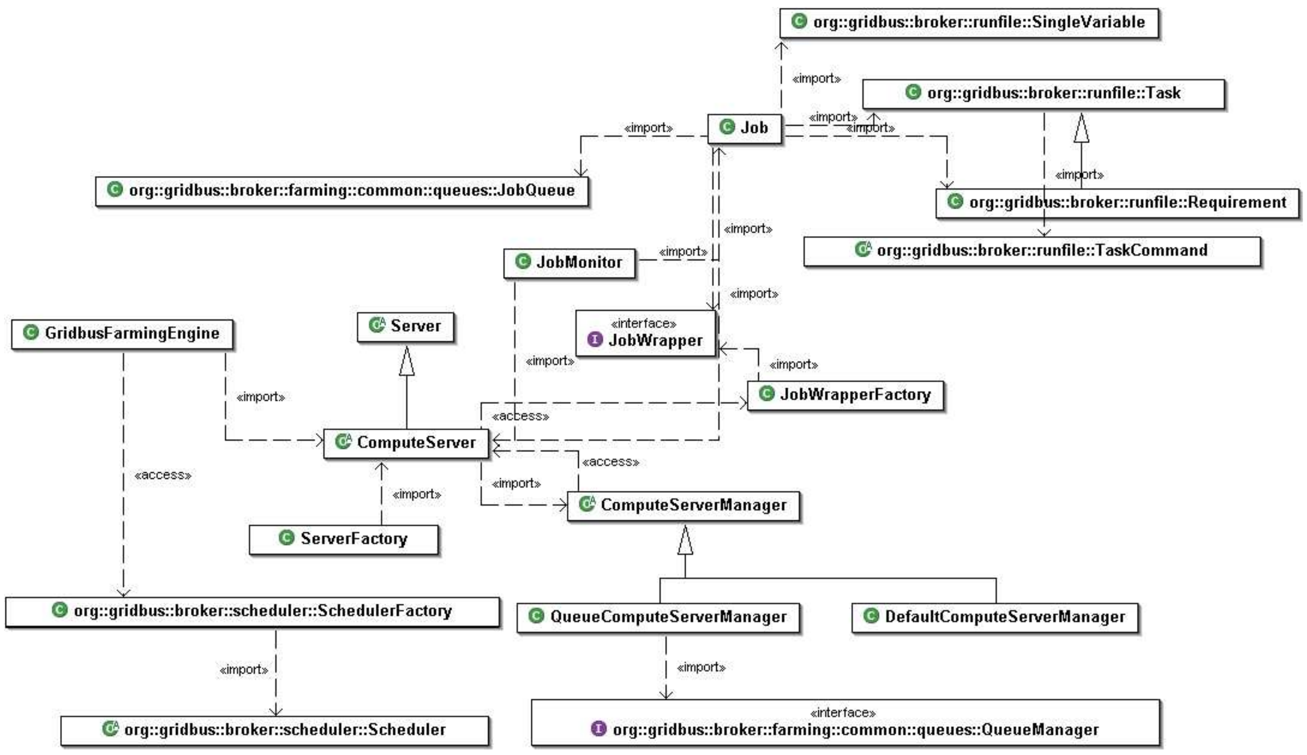

org::gridbus::broker::runfile::SingleVariable
org::gridbus::broker::runfile::Task
Job
org::gridbus::broker::farming::common::queues::JobQueue
org::gridbus::broker::runfile::Requirement
org::gridbus::broker::runfile::TaskCommand
JobMonitor
«interface» JobWrapper
JobWrapperFactory
GridbusFarmingEngine
Server
ComputeServer
ServerFactory
ComputeServerManager
org::gridbus::broker::scheduler::SchedulerFactory
QueueComputeServerManager
DefaultComputeServerManager
org::gridbus::broker::scheduler::Scheduler
«interface» org::gridbus::broker::farming::common::queues::QueueManager
«import»
«access»

Figure 4: Broker UML Diagram.

The broker design is based on the architecture described above. The main entities in the broker are:

- Farming Engine
- Scheduler
- Job
- ComputeServer
- Service

- DataHost and DataFile

The farming engine is the central component which maintains the overall state of the broker at all times. It is the glue that binds all the components together. It acts as a container for the job and server collections. It is the component that interacts with external applications and initiates the scheduling. The farming engine can be considered as the broker's in-memory database, as it holds the broker's current state at any point of time. If using persistence, the farming engine keeps saving its state periodically to a persistence storage.
The scheduler is a middleware independent component which maps jobs to nodes on the grid. It doesn't need to worry about lower level details, and can schedule jobs based on metrics which do not depend on the underlying platform. The broker comes with three built-in schedulers :

- economy-based scheduler (DBScheduler)
- data-aware scheduler (DataScheduler)
- economy and data-aware scheduler (DBDataScheduler)

For more details on the scheduler algorithms please refer to the Gridbus Broker paper at
`http://www.gridbus.org/papers/gridbusbroker.pdf`
A `Job` is an abstraction for a unit of work assigned to a node. As described in previous sections, it consists of `Variables` and a `Task`. A variable holds the designated parameter value for a job. A variable can hold a range of values or a set of values or a single value in which case it is called a single variable. A task is the description of what has to be done by the job.It is composed of a set of commands. There are three types of commands - Copy Command, Execute Command, and Substitute Command. The Copy command instructs the broker to copy a file from the source to the destination. It can be used for either copying the files from the broker host to the remote node or vice versa. A Multiple Copy `(MCopy)` is a special case of the `CopyCommand` which instructs the broker to copy multiple files as described with a wildcard (*, ?, etc.). A `GCopy` command provides the ability to copy from one node to another node without the broker intervening in between. (The `GCopy` is currently supported only by the Globus middleware plug-in.)
The `ComputeServer` class describes a node on the grid and holds the properties for that node, eg. its middleware, architecture, OS etc. It is a middleware independent entity and as been extended for different middleware like Globus, Alchemi, Unicore, Condor, PBS, and also to allow forking jobs on the local node. Each `ComputeServer` has an associated `JobMonitor` and `ComputeServerManager` which are also middleware independent. A `ComputeServer` also has an associated `UserCredential` object, which stores the credentials of the user for accessing services provided by that `ComputeServer`. The `JobMonitor` is responsible for monitoring the execution of all jobs submitted to the remote node corresponding to this `ComputeServer`. The `ComputeServerManager` is the component that manages the job-submission. It uses a local buffer, for this purpose, and when the job needs to be submitted to the remote node, it creates an middleware-specific agent, and sends this agent to the remote node for execution. This is done by invoking the appropriate middle-ware `ComputeServer`. The `ComputeServerManager` has been extended to handle both stand-alone grid nodes, and clusters.
A typical submission and monitoring cycle for a job sent to a Globus node go through the following steps: The Scheduler allocates i.e submits a Job to a ComputeServer. The ComputeServer puts the job in its local job buffer and informs the ComputeServerManager. The ComputeServerManager creates a globus-specific agent - a `GlobusJobWrapper`, and sends it to the remote machine. The `ComputeServerManager` then asks the JobMonitor to include this job in the list of jobs it monitors on that server. Since we are submitting to Globus here, the `ComputeServer` to which the Scheduler allocates the job, would actually

be an instance of the `GlobusComputeServer` class, which extends the ComputeServer class. The job monitor periodically queries job status (by calling the `GlobusComputeServer`), and raises status events for all jobs whose status has changed. When the job is done/failed, the monitor stops monitoring that job.
Data Hosts are nodes on which data files have been stored. These objects store the details of the data files that are stored on them such as their path on the disk and the protocol used to access them. The Data Host objects also maintain a list of the compute resources sorted in the descending order of available bandwidth from the host. Data File objects store attributes of input files that are required for an application such as size and location. A Data File object links to the different Data Hosts that store that file.
Overall, the broker is designed to be a loosely coupled set of components working together. The classes in the broker can be grouped into the following main categories:

- Application-description Interpreters
- Middleware support
- Schedulers
- Persistence providers
- Other support utils and modules

By extending the classes in each group, the broker can transparently support many different app-description formats, schedulers, middleware etc. For more information on how to develop one or more these components please refer to the section ”Modifying or Enhancing the broker to suit your needs”.

5.2 Using the broker in your own applications

This section expands on the brief description given in section 3.2 about how to program the broker and use its services in your own application. Since the main APIs are already described in the sections above, we begin here by looking at some simple examples of the usage of the broker API in a java program. The first step in invoking the broker is to create an instance of the ”Farming Engine”. This can be achieved by the following code snippet:

```
//Create a new "Farming Engine"
GridbusFarmingEngine fe=new GridbusFarmingEngine();
```

This creates the farming engine, sets its configuration properties, and also configures the default logging options. If a broker configuration file (Broker.properties) is found, the configuration specified in that file will be loaded. Otherwise, a default set of values will be loaded. The default values for the broker configuration are shown below:

```
ENV=""
DEADLINE = <1 day>
BUDGET = Long.MAX_VALUE
LOCALDIR = System.getProperty("user.dir")
SCHEDULE = "default"
TEMPDIR = ""
USE_PERSISTENCE = "false"
PERSISTENCE_MODE = "db"
```

```
DB_DRIVER = "com.mysql.jdbc.Driver"
DB_CONNECTIONSTRING = ""
DB_USER = ""
DB_PWD = ""
POLLING_TIME = "10000"
MAX_NUM_FILES = "0"

//default logger properties

//root logger
log4j.rootLogger = "FATAL, RL"
log4j.appender.RL = "org.apache.log4j.RollingFileAppender"
log4j.appender.RL.File = "brokerLogs/rootlogger.log"
log4j.appender.RL.MaxFileSize = "1000KB"
log4j.appender.RL.layout = "org.apache.log4j.PatternLayout"
log4j.appender.RL.layout.ConversionPattern = "%p %t %c - %m%n - (%F:%L)"

//Broker-specific logs
log4j.logger.org.gridbus.broker = "DEBUG, stdout, DB"
log4j.logger.org.gridbus.broker.test = "DEBUG,stdout"
log4j.logger.org.gridbus.broker.persistence = "INFO"
log4j.logger.org.gridbus.broker.xpml = "INFO"

log4j.appender.stdout = "org.apache.log4j.ConsoleAppender"
log4j.appender.stdout.layout = "org.apache.log4j.PatternLayout"
log4j.appender.stdout.layout.ConversionPattern = "%5p (%F:%L) - %m%n"
log4j.appender.DB = "org.apache.log4j.RollingFileAppender"
log4j.appender.DB.File = "brokerLogs/broker.log"
log4j.appender.DB.MaxFileSize = "1000KB"
log4j.appender.DB.MaxBackupIndex = "10"
log4j.appender.DB.layout = "org.apache.log4j.PatternLayout"
log4j.appender.DB.layout.ConversionPattern = "%d %p [%t] (%F:%L) - %m%n"
```

After creating the farming engine, the next step is to setup the jobs and servers. This can be done in two ways. One way is to create jobs from an application description file (currently only the XPML format is supported). The other way to create jobs is to use the `Task` and `Command` APIs which give the programmer more flexibility. Similarly servers can be setup using a resource list file supplied to the farming engine, or using the `ServerFactory` APIs. To setup jobs and servers using an application and resource description files use the following:

```
fe.setAppDescriptionFile("calc.xml");
fe.setResourceDescriptionFile("resources.xml");
fe.init();
```

[Note: Use the init method of the farming engine only when supplying the application- and resource- description files. The init method fails if these files are not given. The init method itself calls the two methods - initResources() and initJobs(), in that order. This order is to be maintained strictly if the initResources() / initJobs() methods are explicitly called by another program. This is because the initResources() sets up the broker with services which are queried, for example, to find out the datahosts and datafiles needed by the jobs in an application.]

To use the APIs to create jobs:

```
Job currentJob = new Job();
currentJob.setJobID("j1");

//Create commands

//Command to Copy the program
CopyCommand copy = new CopyCommand();
copy.setSource(true,"workingDIR/calc",true);
copy.setDestination(false,"calc",true);

//Command to execute the program
ExecuteCommand exec = new ExecuteCommand(TaskCommand.EXECUTE_CMD);
exec.setExecutable("./calc");
exec.addArgument("$X");
exec.addArgument("$time_base_value");

//Command to collect the results
CopyCommand results = new CopyCommand();
results.setSource(false,"output",true);
results.setDestination(true,"output."+currentJob.getJobID(),true);

Task task = new Task();
task.addCommand(copy);
task.addCommand(exec);
task.addCommand(results);

currentJob.setTask(task);

currentJob.addVariable(new SingleVariable("$X", "1"));
currentJob.addVariable(new SingleVariable("$time_base_value", "0"));

fe.addJob(currentJob);
```

First, a new `Job` object is created. A job object is associated with `Variables` and a `Task` which has any number of commands which can have commands such as `Copy,GCopy,MCopy,Execute,and Substitute`. For copy commands, `(Copy,MCopy,GCopy)`, the source and destination file names need to be set. The `ExecuteCommand` class can be used to execute programs on the remote node. As shown above, the executable and arguments need to be set for the `ExecuteCommand`. Substitute commands are used to substitute variables in strings or text files locally, before sending the job to the remote side. Variable objects can be created and added to a job as shown. Variables are normally added as strings (even though they are numeric). The variable values can also refer to filenames. The XPML interpreter has certain file-resolver APIs which are used to resolve the physical filenames from those containing wildcard characters. These need to be used, so that the `DataHosts` and `DataFiles` collections used in data-aware scheduling

are properly set up. (Please refer to the javadocs for further details on their usage.) Finally, after the task is set and variables are added, this job is added to the list of jobs in the farming engine. The broker retrieves the stdout and stderr of a job by default. Any other output files generated by the job, however need to be explicitly copied back to the broker node(i.e the machine running the broker). Alternatively, the job outputs maybe copied to some other remote location by the job as the last step after execution.
Servers are created using the `getComputeServer` method of the `ServerFactory` class. This creates a middleware-specific compute-server and returns it in the form of an instance of the generic `ComputeServer` abstract class. The `ServerFactory` has various overloaded methods used to create a `ComputeServer` instance. The least information needed is the hostname of the `ComputeServer`. If the "type" of compute-server is not specified, it is assumed to be a Globus compute-server. The (middleware) type of a compute-server is any one of the supported middleware, currently: `GLOBUS,ALCHEMI,UNICORE,FORK,CONDOR,PBS`. The following code snippet shows the creation of a compute-server:

```
      LocalProxyCredential lpc = new LocalProxyCredential();
      lpc.setPassword("somepassword");

      ComputeServer cs = ServerFactory.getComputeServer(ComputeServer.GLOBUS,
"belle.cs.mu.oz.au");
      cs.setUserCredential(lpc);
      fe.addServer(cs);
```

The first two lines of code, set the credentials for the server. This can be avoided if using a resource description file, in which the credentials are specified. When a compute-server is created, the server is not alive, and the `isAlive` method returns false. The compute-server needs to be started (in order to facilitate the discovery process), before any of its methods can be put to good use. This can be done by calling the "start" method. The start method initiates the compute-server manager, and a job-monitor to keep track of job submission and status-updating respectively. The manager performs an initialization of the compute-server properties by "discovering" them. It does so by pinging the remote node, then connecting to it and querying it for certain information. Once the properties are discovered, the `isAlive` flag is set to true. This whole process is done on a seperate thread for each server (i.e the start method returns immediately). However, the farming engine makes sure the start method is called on all servers before starting scheduling. So, in effect it is not necessary to call this method explicitly if the jobs are to scheduled to servers.
To begin scheduling jobs, the schedule method needs to be called on the farming engine. This will get the scheduler defined in the Broker.properties file, if it exists, or the "default" scheduler otherwise. There are two ways to override this behaviour. One of them is to set the `SCHEDULE` property of the `BrokerProperties` class. The possible values for this are- cost,time, costtime, costdata, timedata, default. The alternate method is to explicity set the scheduler for the farming engine using the `SchedulerFactory` class. The following code shows how to do this:

```
    //1. simply call the schedule method, to invoke the default scheduler
    //the scheduler defined in the BrokerProperties
    fe.schedule();

    //OR
```

```
//2. set the scheduling type / algorithm in the BrokerProperties
//before calling the schedule method.
//for cost-optimizing scheduler...
BrokerProperties.setProperty("SCHEDULE","cost");
fe.schedule();

//OR

//3. explicitly create a scheduler object
//and set the farming-engine's scheduler
Scheduler s = SchedulerFactory.getScheduler("cost",fe);
fe.setScheduler(s);
fe.schedule();
```

The schedule method of the farming engine starts the scheduler on a seperate thread, and returns immediately. The broker provides built-in mechanisms for job-status-monitoring and updating of job statistics such as active-jobs, completed-jobs, failed-jobs etc. Once the scheduling is started, the jobs get submitted to remote nodes, and as their status changes the `JobMonitor` for each server updates the status and the scheduler updates the statistics. The `JobMonitor` uses events to signal `JobListeners` about `statusChanged` events for jobs. To receive job-status events the `JobListener` interface needs to be implemented.
The broker has a built-in logging system, that uses the Apache log4j APIs. When the broker is used via the command-line or the APIs, it creates a log file called broker.log in a temporary directory created for each run of the broker. The temporary directory is of the form `DBxxxxxxxxxxxxxx` (where the x's are obtained from the current date-time stamp). In this temp directory are all the files generated by the broker and the jobs it schedules. The broker logs are useful for reviewing the broker output messages and debugging. Each job produces an `stdout.<jobid>` and a `stderr.<jobid>` file containing the standard output and error. There is also one shell file per job, in the directory. This is the file that actually gets executed on the remote node. Apart from these files there are also any other outputs / files generated/copied back by the job itself. On the remote node, each invocation of the broker creates a temporary directory that looks like `GB2.<brokerID>`. (where `<brokerID>` is the unique ID of the broker generated for each run). Inside this directory is one directory for each job that gets submitted to the remote node. Inside each job's temp directory on the remote node, are all its files - inputs and outputs. Normally, the broker deletes the temp directories on the remote node after it completes execution on the node. The shell files on the local side are also deleted. However, if the logger is set to DEBUG level, then these shell files and remote directories are left alone, to assist in debugging.
This section has hopefully helped you to gain a reasonable understanding of the broker APIs to get started on working with the broker. You are encouraged to have a look at the full java-docs for the broker at `http://www.gridbus.org/broker/2.0/docs/`

5.3 Using the broker in a web portal (portlets)

The previous section provided in-depth instructions on how to integrate the broker into Java applications by making use of the broker API. This section will build on that knowledge and walk through an example of how the broker could be used within a portlet environment, highlighting any interesting details along the way. This guide assumes the programmer has some experience working with a portlet framework (such as

Gridsphere) and with portlet development.

5.3.1 What are portlets?

Portlets are reusable Web components that are used to compose web portals. A news oriented portal may make use of a news article portlet, a weather portlet, a stock quote portlet and maybe a search portlet. These portlets may be completely independent but are organised and presented to the user together as a single web page. Portlet standards such as `JSR168` enable developers to create portlets that can be plugged into any portal supporting those standards.

5.3.2 Why portlets are good for us

Portlets are now becoming a very popular and prominent new technology. With the promotion of reusability and portability (between different portlet frameworks) through standards (`JSR168`), there is definitely a strong case for the use of portlets. A large number of portlet frameworks are also available to meet varying demands.
Within a community of web portal developers with a desire for code sharing, the case for portlets just gets stronger.

5.3.3 The Gridbus Broker portlets

These Gridbus Broker portlets show off some of the broker's features. The aim is to both provide programmers with an example of what is possible with the broker so that much more advanced portlets can then be developed, as well as providing the community with a set of reusable portlets providing sophisticated yet simple Grid application management and execution, that can be incorporated into any web portal.
The aim of this guide is to help deploy the example portlets and try to get them to run with Gridsphere portlet framework. The portlets are `JSR168` compliant, so even though they were tested within Gridsphere, they should be portable to other frameworks.

5.3.4 Pre-requisites

This guide assumes that the programmer has a working installation of Gridsphere `(http://www.gridsphere.org).`
This means that you'll also have an installation of Jakarta Tomcat `(http://jakarta.apache.org/tomcat).`
If this has not already done, it should be installed before continuing.
It should be noted that due to the need for GSI security, the machine running the portal needs to know about the CA certificates of any Grid resources that will be used. Globus cog will look for certificates in the .globus directory under the current user's user directory (both windows or *nix). Make sure all required CA certificates are located in the appropriate place taking into consideration the user under which Gridsphere is executing as. In the case of the author, who is running Gridsphere in Windows XP, the directory was:

```
C:\WINDOWS\system32\config\systemprofile\.globus\certificates
```

On a Windows 2000 machine, this could be something like:

```
C:\Documents and Settings\Default User\.globus\certificates
```

Under *nix it would be something like

```
/home/hag/.globus/certificates
```

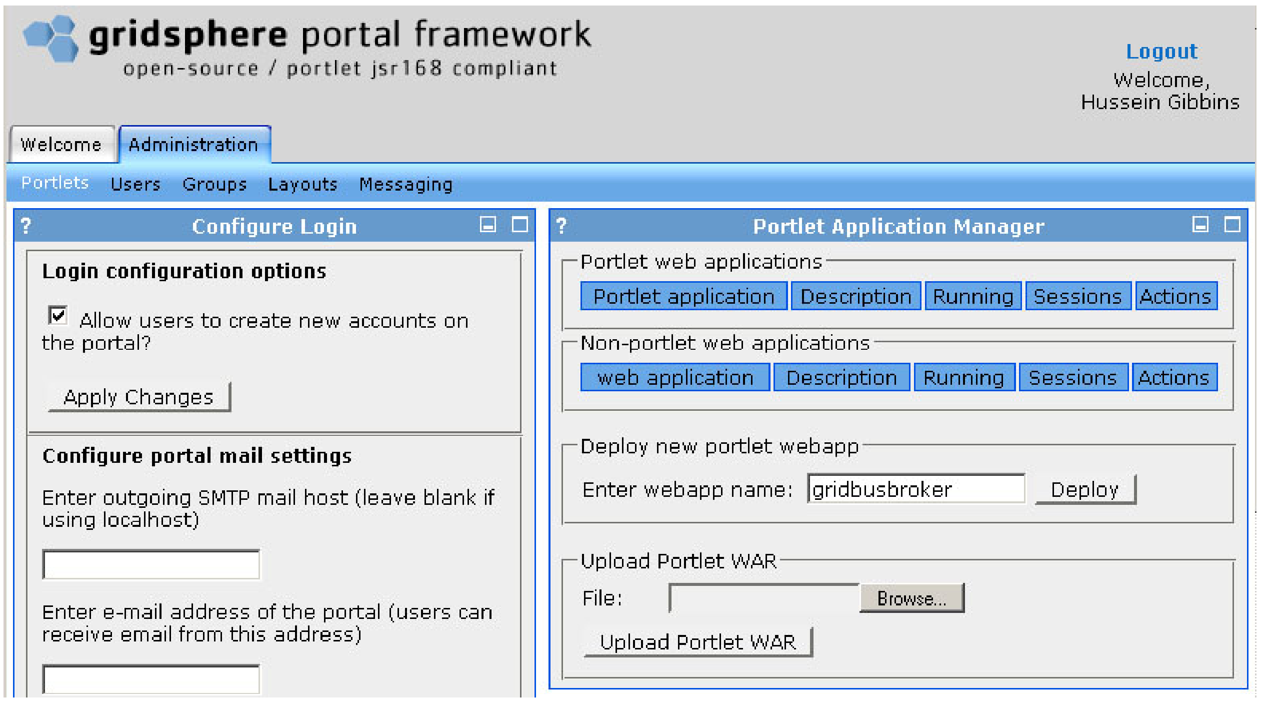
5.3.5 Deploying and installing the broker portlets (with Gridsphere)

Next, deploy the set of portlets according to the instructions for the portlet framework being used. Two ways you can do this in Gridsphere are described below:
[Note: If you have downloaded the source version of Gridsphere and have deployed Gridsphere into a pre-existing Tomcat installation, `$CATALINA_HOME` will be the path to Tomcat. If you installed Gridsphere packaged with Tomcat, then `$CATALINA_HOME` is the path to the Gridsphere installation.]

- Deploying the web archive
  - Copy the `"gridbusbroker.war"` web archive into the `$CATALINA_HOME/webapps` directory.
  - Restart Tomcat and the web archive will be automatically deployed.
  - Login to the Gridsphere portal and access the ”Portlet Application Manager” portlet, which is located under the ”Portlets” submenu of the ”Administration” menu.
  - Deploy the gridbusbroker portlet webapp as shown in Fig. 5 By entering ”gridbusbroker” in the ”Deploy new portlet webapp” text field and clicking ”deploy”, Gridsphere becomes aware of the new portlets.
  - Create a new group and add the gridbusbroker portlets to it. To access this option go to the ”Administration” menu, then to the ”groups” submenu, and then select the ”create new group” option. Fig.6
  - Tell Gridsphere to create a Template layout for the portlets.
  - Go to the ”Welcome” menu and tick the box next to the gridbusbroker portlet webapp and then save, as shown in Fig.7. A new menu will be displayed that will give you access to the gridbusbroker portlet webapp.
- Deploying the Gridsphere project using ”ant deploy”
  If you have downloaded the source version of Gridsphere you can alternatively deploy the Gridsphere project version of the gridbusbroker portlets.
  - Start by extracting `gridbusbroker_project.tar` into the projects sub-directory of your Gridsphere source installation. The projects directory gets created when you run ”ant new-project” from the Gridsphere source installation directory. If the projects directory doesn’t already exist, simply create it. The basic directory structure should look as follows:

```
<path to Gridsphere source>/
                        /projects/
                            /gridbusbroker/
                                /build/
                                /config/
                                /lib/
                                /META-INF/
                                /src/
                                /webapp/
                                build.properties
                                build.xml
```

  - Enter the `/projects/gridbusbroker/` directory


gridsphere portal framework
open-source / portlet jsr168 compliant

Logout
Welcome,
Hussein Gibbins

Welcome | Administration

Portlets | Users | Groups | Layouts | Messaging

Configure Login

Login configuration options

[✓] Allow users to create new accounts on the portal?

Apply Changes

Configure portal mail settings

Enter outgoing SMTP mail host (leave blank if using localhost)

Enter e-mail address of the portal (users can receive email from this address)

Portlet Application Manager

Portlet web applications

Portlet application	Description	Running	Sessions	Actions

Non-portlet web applications

web application	Description	Running	Sessions	Actions

Deploy new portlet webapp

Enter webapp name: gridbusbroker Deploy

Upload Portlet WAR

File: Browse...

Upload Portlet WAR

Figure 5: Deploy Broker Portlets

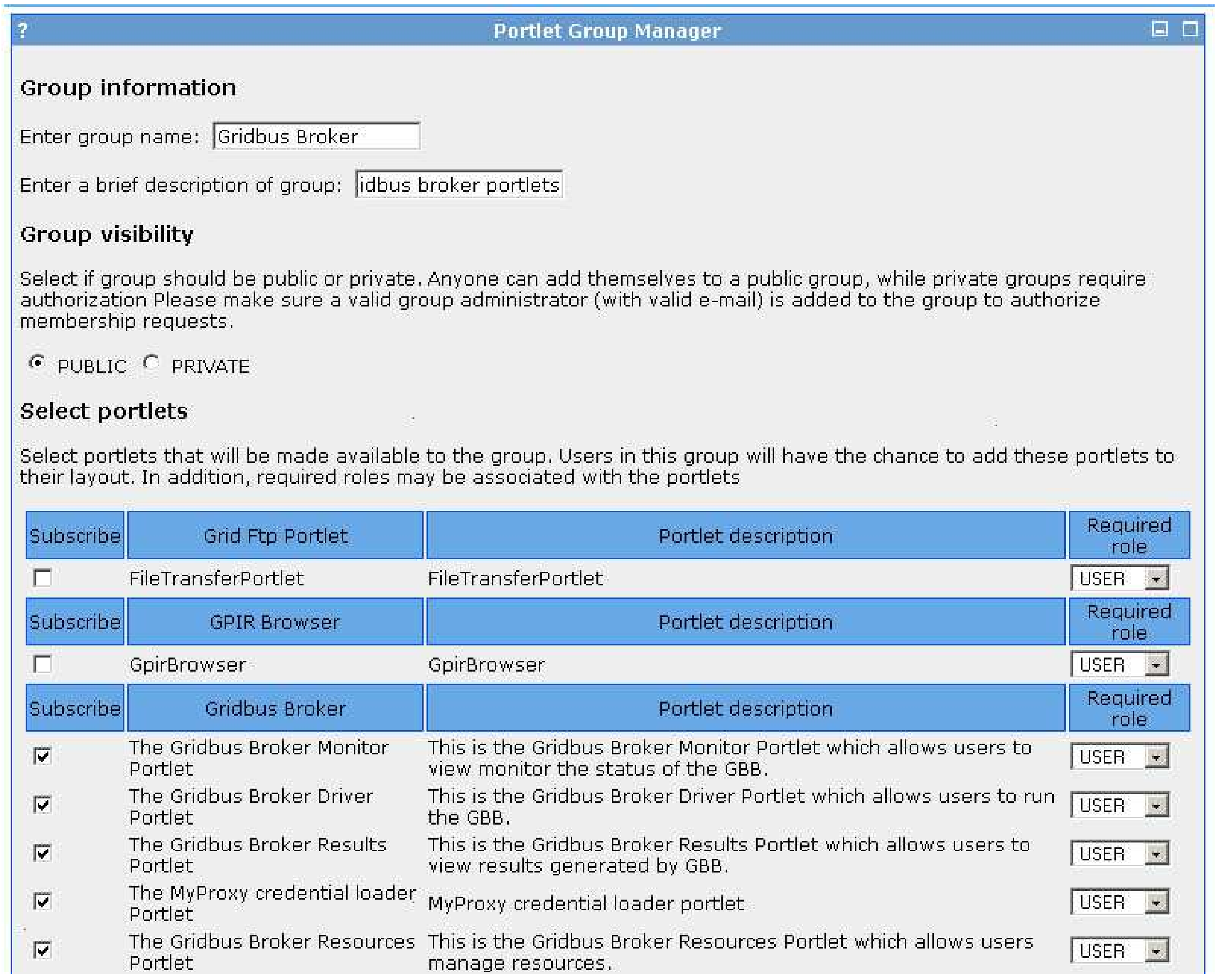
- Ensure `$CATALINA_HOME` is correctly pointing to your Tomcat installation and run the command ”ant deploy”. This will deploy the gridbusbroker portlets.
- Restart the Tomcat server and then follow steps (starting from the third step) from ”Deploying the web archive”.

Now that the broker demo portlets are accessible, it is time to see what each of them does and how they have been implemented!

5.3.6 The MyProxy Portlet

`(org.gridbus.broker.examples.portlets.gridbusbroker.GetMyProxy.java)`

By default a user will not be authorised to interact with any of the broker portlets yet, this is because they need a valid proxy so that work can be done out on the Grid (assuming the user wants to execute on Globus nodes). So initially the user will only have access to the MyProxy portlet. The MyProxy portlet will allow users to specify the location of a MyProxy server as well as the username and password of a proxy stored at that location. The portlet will retrieve the proxy and make it available to the rest of the portlets by putting it into the session (note that the object must be put into application scope or else it won’t be found by other portlets). This proxy, or user credentials, will be later passed to the broker in order for it to authenticate to other Grid resources.



Group information

Enter group name: Gridbus Broker

Enter a brief description of group: idbus broker portlets

Group visibility

Select if group should be public or private. Anyone can add themselves to a public group, while private groups require authorization Please make sure a valid group administrator (with valid e-mail) is added to the group to authorize membership requests.

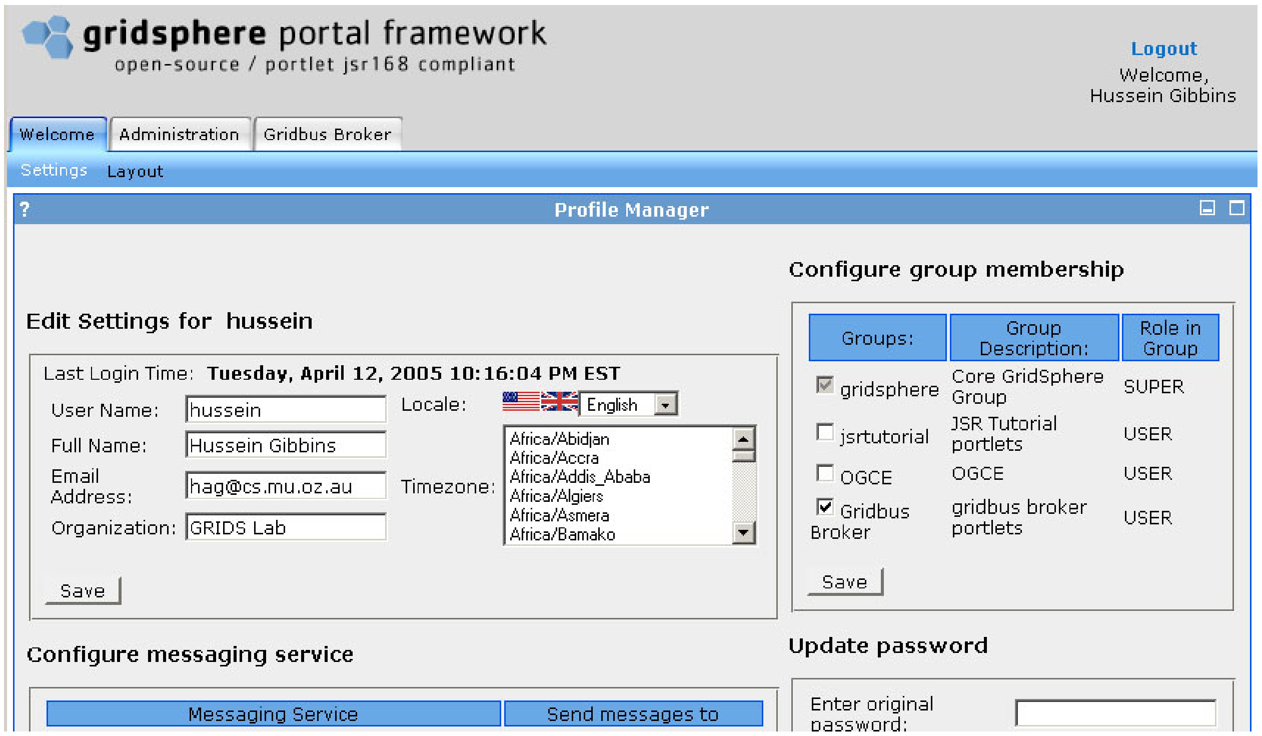
[✓ PUBLIC] PRIVATE

Select portlets

Select portlets that will be made available to the group. Users in this group will have the chance to add these portlets to their layout. In addition, required roles may be associated with the portlets

Subscribe	Grid Ftp Portlet	Portlet description	Required role
☐	FileTransferPortlet	FileTransferPortlet	USER
Subscribe	**GPIR Browser**	**Portlet description**	**Required role**
☐	GpirBrowser	GpirBrowser	USER
Subscribe	**Gridbus Broker**	**Portlet description**	**Required role**
☑	The Gridbus Broker Monitor Portlet	This is the Gridbus Broker Monitor Portlet which allows users to view monitor the status of the GBB.	USER
☑	The Gridbus Broker Driver Portlet	This is the Gridbus Broker Driver Portlet which allows users to run the GBB.	USER
☑	The Gridbus Broker Results Portlet	This is the Gridbus Broker Results Portlet which allows users to view results generated by GBB.	USER
☑	The MyProxy credential loader Portlet	MyProxy credential loader portlet	USER
☑	The Gridbus Broker Resources Portlet	This is the Gridbus Broker Resources Portlet which allows users manage resources.	USER

Figure 6: Create a new group for Broker Portlets



gridsphere portal framework
open-source / portlet jsr168 compliant

Logout
Welcome,
Hussein Gibbins

Welcome | Administration | Gridbus Broker

Settings | Layout

Profile Manager

Edit Settings for hussein

Last Login Time: **Tuesday, April 12, 2005 10:16:04 PM EST**

User Name:	hussein
Full Name:	Hussein Gibbins
Email Address:	hag@cs.mu.oz.au
Organization:	GRIDS Lab
Locale:	English
Timezone:	Africa/Abidjan, Africa/Accra, Africa/Addis_Ababa, Africa/Algiers, Africa/Asmera, Africa/Bamako

Save

Configure group membership

Groups:	Group Description:	Role in Group
✓ gridsphere	Core GridSphere Group	SUPER
☐ jsrtutorial	JSR Tutorial portlets	USER
☐ OGCE	OGCE	USER
✓ Gridbus Broker	gridbus broker portlets	USER

Save

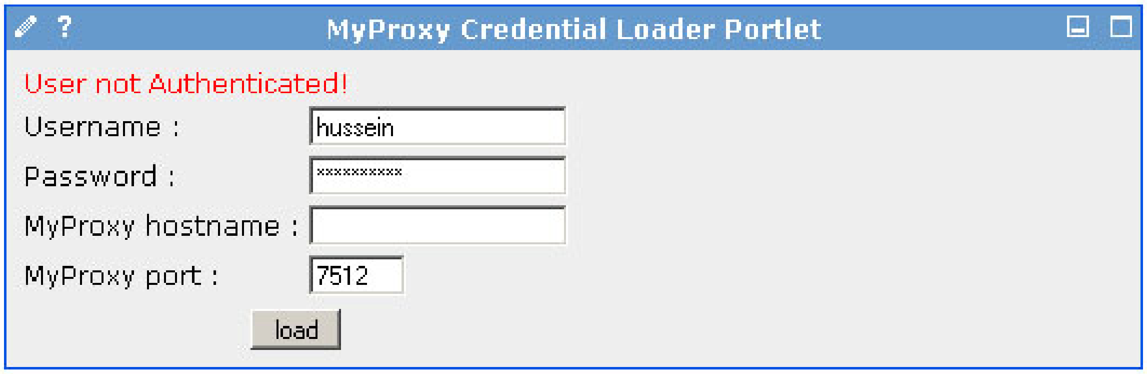
Configure messaging service

Messaging Service	Send messages to

Update password

Enter original password:

Figure 7: Configure Broker Portlets group


MyProxy Credential Loader Portlet

User not Authenticated!

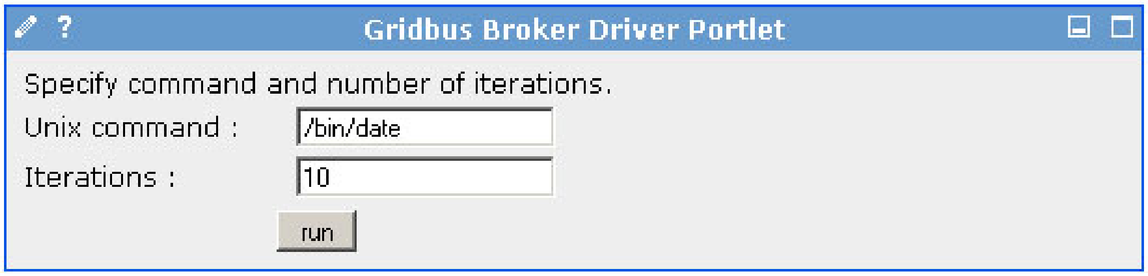
Username :	hussein
Password :	**********
MyProxy hostname :	
MyProxy port :	7512

Figure 8: MyProxy Portlet

```
  GSSCredential cred = null;
  MyProxy myproxy = null;

//   connect to myproxy server.
  myproxy = new MyProxy(host, Integer.parseInt(port));
//   get the credential for this user.
  cred = myproxy.get(username, password, 3600);
//   store credential in session in application scope so other portlets can use it.
  session.setAttribute("userCred", cred, PortletSession.APPLICATION_SCOPE);
```

5.3.7 The Gridbus Broker Driver Portlet

(org.gridbus.broker.examples.portlets.gridbusbroker.GBBDriver.java)


Gridbus Broker Driver Portlet

Specify command and number of iterations.

Unix command :	/bin/date
Iterations :	10

Figure 9: GBB Driver Portlet

The driver portlet is where the majority of the code which uses the broker API is. Here the broker is initialized and told to start it executing on the Grid. The portlet is simple - it allows the user to specify a Unix command and the number of times that they wish to execute that command. Running commands like `/bin/hostname` or `/bin/date` are usually the most interesting, as they tell the user where or when the command was executed.
Once the command is submitted, the portlet needs to:

Create an instance of the farming engine

```
//   create a new farming engine if one doesnt already exist
  GridbusFarmingEngine fe = (GridbusFarmingEngine) session.getAttribute("fe");
  if (fe == null) {
    try {
     fe = new GridbusFarmingEngine();
    } catch (Exception e) {}
    session.setAttribute("fe",fe,PortletSession.APPLICATION_SCOPE);
  }
```

Create the jobs (as described in other sections of the manual) according to the command and number of iterations specified.

```
  Vector jobs = null;
  //  ...
  //create the jobs based on the command and number of iterations
  //specified by the user.
  jobs = createJobs(command, Integer.parseInt(iterations));
```

Add the jobs to the farming engine and start the broker.

```
  fe.setJobs(jobs);
  //schedule our jobs for execution on the Grid!
  fe.schedule();
```

It should be obvious that when creating the jobs, the programmer could implement the `createJobs()` method to create jobs that do anything that the broker would usually be able to handle.
When referring to any files locally, such as an executable that needs to be copied and executed on a remote node, it may be handy to make use of the

```
   PortletContext.getRealPath(java.lang.String path)
```

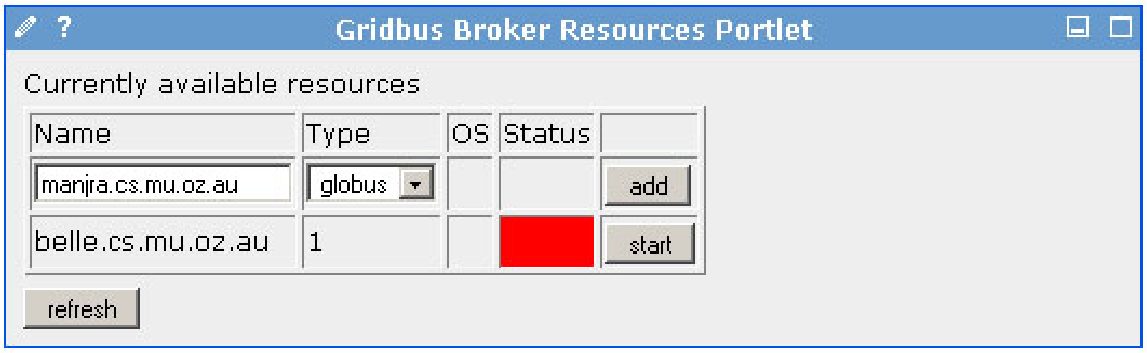
method This method will help avoid hard-coding absolute paths in portlets.
Well now the broker is running and the user's command is being run over the Grid...well not exactly. The broker doesn't know what resources to use yet. Luckily there is a portlet that allows the user to specify their resources.

5.3.8 The Gridbus Broker Resources Portlet

```
 (org.gridbus.broker.examples.portlets.gridbusbroker.GBBResources.java)
```

The job of this portlet is to tell the broker which resources to run on. Here the user can add any resources they want to be a part of their ”Grid”. Remember that the proxy will need to be valid on any resource that is identified in order for the broker to be able to access it on the user's behalf. When a new resource is submitted by the user, the portlet needs to make sure there is an instance of the farming engine or else there will be nothing to add the resource to. Doing this allows resources to be added before starting execution via the Driver portlet:

```
  if (fe == null) {
    try {
```


Gridbus Broker Resources Portlet

Currently available resources

Name	Type	OS	Status	
manjra.cs.mu.oz.au	globus			add
belle.cs.mu.oz.au	1		🟥	start

Figure 10: GBB Resources Portlet

```
        fe = new GridbusFarmingEngine();
    } catch (Exception e) {}
    session.setAttribute("fe",fe,PortletSession.APPLICATION_SCOPE);
}
```

Since the broker is already running, resources are now being added on the fly! Adding resources to the broker is simple and is detailed elsewhere in the manual. Within a portal it is the same - just get the hostname from the form and add it to the farming engine. Note that the default compute server type is globus, and the ProxyCredential is given the credential that was previously retrieved from MyProxy.

```
if (req.getParameter("type").compareTo("globus") == 0) {
    ProxyCredential pc = new ProxyCredential();
    pc.setProxy((GSSCredential) session.getAttribute("userCred"));
    ComputeServer cs = ServerFactory.getComputeServer(req.getParameter("host"));
    fe.addServer(cs);
}
```

When adding resources prior to starting the broker with a `farmingEngine.schedule()`, the call to schedule will automatically initialize and start all available resources. If the resources have been added after scheduling has started (as with the path of this guide) each resource will have to be started manually.
When a resource is added, the portlet will show that the resource is in the ”stopped” state and there will be a ”start” button beside it. The user must click on the ”start” button to start the resource. Starting the resource will allow the broker to identify the resource as being active and will then start giving it work to do. If an invalid resource is supplied, the startup will fail.

```
//get the current set of compute servers and set the selected server to alive
Collection resources = fe.getServers();
ComputeServer cs = null;
for (Iterator it=resources.iterator();it.hasNext();) {
  cs = (ComputeServer)it.next();
  if (cs.getServerID().compareTo(req.getParameter("serverID")) == 0) {
    cs.startup();
    break;
  }
}
```

This loop finds which resource needs to be started, and then calls the compute server's startup method. The portlet may return before the resource's status has been updated, so a refresh may be required to update the display. A resource can also be stopped during execution. The implementation is the same, but the shutdown method is called instead of startup.
Another way to specify to the broker which resources it should use is by using the resource list file (as discussed earlier in the manual). One could imagine allowing users to upload a resource list and then passing that onto the broker. Another option would be to read a resource list that already exists on the server's file system. There is an example resource list in the directory:

`gridbusbroker\webapp\files\config\resources.xml`

Here is code that can be used to load that file.

```
//load the description of compute servers from a file.
String resourceDesc = getPortletContext().getRealPath("/files/config/resources.xml");
fe.setResourceDescriptionFile(resourceDesc);
```

Of most interest here is the following call

`getPortletContext().getRealPath("/files/config/resources.xml")`

This allows the programmer to refer to resources contained within the web application's directory structure. Otherwise Java may look in some other location on the system.

5.3.9 The Gridbus Broker Monitor Portlet

`(org.gridbus.broker.examples.portlets.gridbusbroker.GBBMonitor.java)`

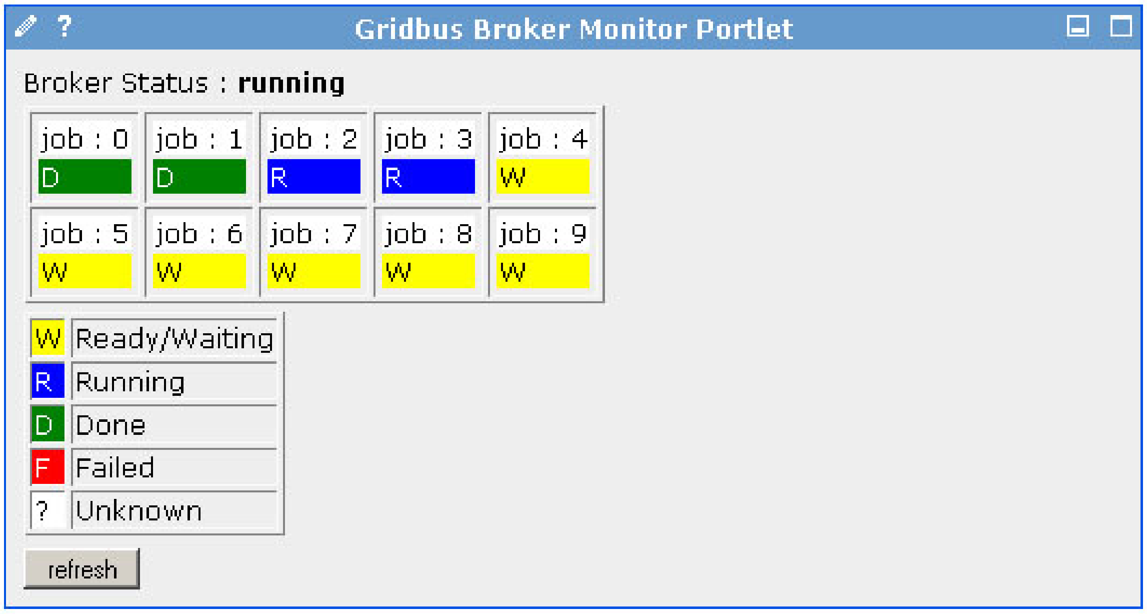


Figure 11: GBB Monitor Portlet

The broker API provides the programmer with the ability to get all of the jobs and query their status information. Using this, a simple monitor can be created to report the status of each of the jobs being run.

```
Collection jobs = new Vector();
jobs = fe.getJobs();
```

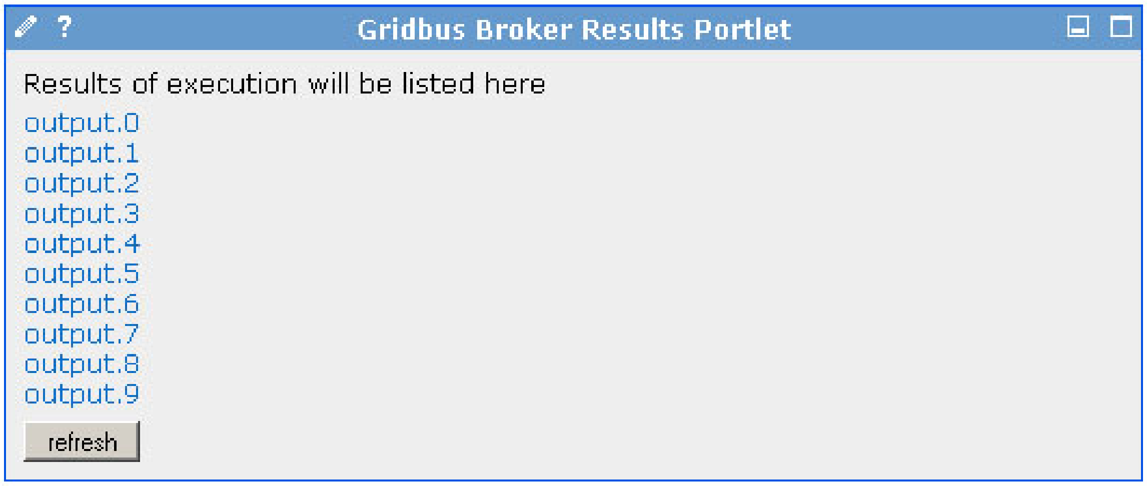
The collection of the jobs then needs to be sent to the view (the JSP) for formatting, where each job will be queried for its status. The broker's overall status can also be queried to determine whether it is stopped, running, failed, finished or complete.

```
//check the overall status of the broker
if (fe.isSchedulingFailed()) {
  status = "failed";
} else if (fe.isSchedulingFinished()) {
  status = "finished";
} else {
  status = "running";
```

More advanced things can be done, like checking which server is executing which job, cancel jobs, restart any jobs which may have failed and so on. What the programmer decides to do with all the information and flexibility provided by the broker is left to the imagination.
Now the broker has finished executing, but how can the user really tell that the commands were successfully run on the remote nodes?

5.3.10 The Gridbus Broker Results Portlet

```
(org.gridbus.broker.examples.portlets.gridbusbroker.GBBResults.java)
```



Results of execution will be listed here
output.0
output.1
output.2
output.3
output.4
output.5
output.6
output.7
output.8
output.9

Figure 12: GBB Results Portlet

This portlet displays, and allows the user to download, the stdout generated by each of the iterations. The content of output.4 is shown below:

```
*This job is running on host: belle.cs.mu.oz.au
*Executing command: /bin/date Wed Apr 13 01:10:56 EST 2005
GBX:0
*Command done
*Copying
file:///home/bellegrid/GB2.6AC7C2AB-FFC0-E7AE-A4E8-ED23BACEB122/DB1113318673296.4/stdout.4
to https://128.250.34.16:3259/brokerLogs/DB1113318391140/output.4
The destination is
https://128.250.34.16:3259/brokerLogs/DB1113318391140/output.4
```

[Note: On windows there is a known issue with the broker having trouble copying files to an absolute path locally. This is due to globus GASS using https URIs and the inability to specify a drive letter as part of that URI in windows. Because of this, the outputs are copied over from the broker's working directory into a directory where they can be accessed by the web portal whenever the results portlet is refreshed.]
First the source and destination for the copy need to be set:

```
//move results from brokers temp directory to results directory
File tempDir = new File(fe.getTempDirectory());
File resultsDir = new File(getPortletConfig().getPortletContext().getRealPath(
            "/files/results")+"/"+fe.getBrokerID());
```

The results are then copied into a unique directory based on the broker's ID. This way, each run will put results in a new directory.

```
resultsDir.mkdirs();
File[] files = tempDir.listFiles();
for (int i = 0; i < files.length; i++) {
  if (files[i].getName().startsWith("output")) {
  File newFile =
    new File(resultsDir.getAbsolutePath()+"/"+files[i].getName());
        files[i].renameTo(newFile);
  }
}
```

Now all the results will be copied over to a directory relative to the web portal so they can be easily accessed. The only problem is that the copying only happens when the portlet is reloaded. This means that if the portlet is never reloaded then the files will never actually be copied. This is just one solution to the problem. Another solution could involve starting a new thread to monitor the broker's output directory and copy results as soon as they appear. Programmers should also be aware that any output from an execution (output files from an executable) can be accessed like this, not only the stdout.
Well that's it! This is just an example of what is possible! Hopefully this has demonstrated how easily the broker can be integrated into portals. To Grid-enable a legacy application with a portlet interface, the programmer needs to just modify the Driver and Results portlets to suit the needs of the application.

5.4 Modifying or Enhancing the broker code to suit your needs

The Gridbus broker is an open-source effort, and any help from the community to improve the broker / fix bugs / add new features etc. is always welcome. Contributions / enhancements to the broker code may be included in future distributions, after being tested by the broker development team. The v.2.0 distribution ships with all the source code, and this section describes the design main groups of APIs from the point of view of a developer wanting to contribute towards the broker development. The following information applies to those who want to modify the broker source code to suit their needs for example, write custom schedulers, middleware plug-ins, application-description interpreters etc for the broker.

5.4.1 Application Interfaces / Application Description Interpreter APIs

The first step before running any grid-application would be to describe it in some form. This step basically involves capturing the requirements and codifying them in a format that can be interpreted by the broker. The design of the broker allows it to work with different application descriptions, so long as there are appropriate ”interpreters” which convert the description into objects that the broker works with, i.e the application description is mapped to the broker’s core objects such as jobs, servers, datahosts, datafiles etc. The default application description system, based on XML is provided with the broker is XPML (Extensible Parametric Modeling Language). The XPML schema is a bit texty and lengthy, so a graphical representation is shown in Figs. 13,14 and 15

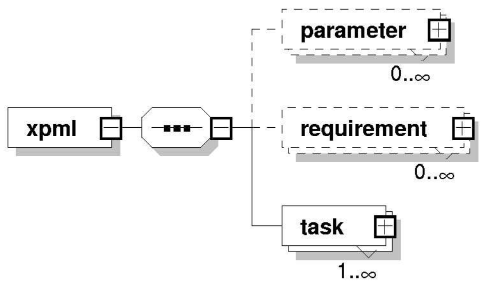


Figure 13: Broker XPML Schema Diagram

The entire schema is not yet fully implemented in the broker. The XPML interpreter is designed to be flexible so that changes in the schema are easily accommodated. It makes extensive use of the Java reflection API. The entry point into the XPML interpreter is the `parseXML` method of the `XMLReader` class. This loads an Interpreter class by reflection, and calls its process method. In this case the `XpmlInterpreter` is loaded. The interpreter processes the parameters by calling the `ParameterProcessor` class again using reflection, which in turn reflects on each element that occurs inside a ¡parameter¿ element. The idea here is to be easily able to add extensions to the schema, while minimizing the changes that need to be made to the interpretation process. So the `ParameterProcessor` can call one of `IntegerProcessor, StringProcessor, GridfileProcessor` etc. After all the parameters are parsed, the `XpmlInterpreter` parses the requirements and task, using the `RequirementProcessor` and `TaskProcessor`. The `TaskProcessor` also uses reflection to parse each Command.
So, methods like `processCopy, processExecute, processGCopy` etc are called by reflection. This makes it easy to add a new type of command to the schema, and all that is needed would be to add a new `processXXXXX` method to the `TaskProcessor` to handle the new command. The parameters are tasks parsed are then converted into variables and added to jobs which are created by the `XpmlInterpreter`.
To make the broker work with different application description schemes, an interpreter for the particular description needs to be written. The interpreter would ideally provide methods similar to the `XMLReader`

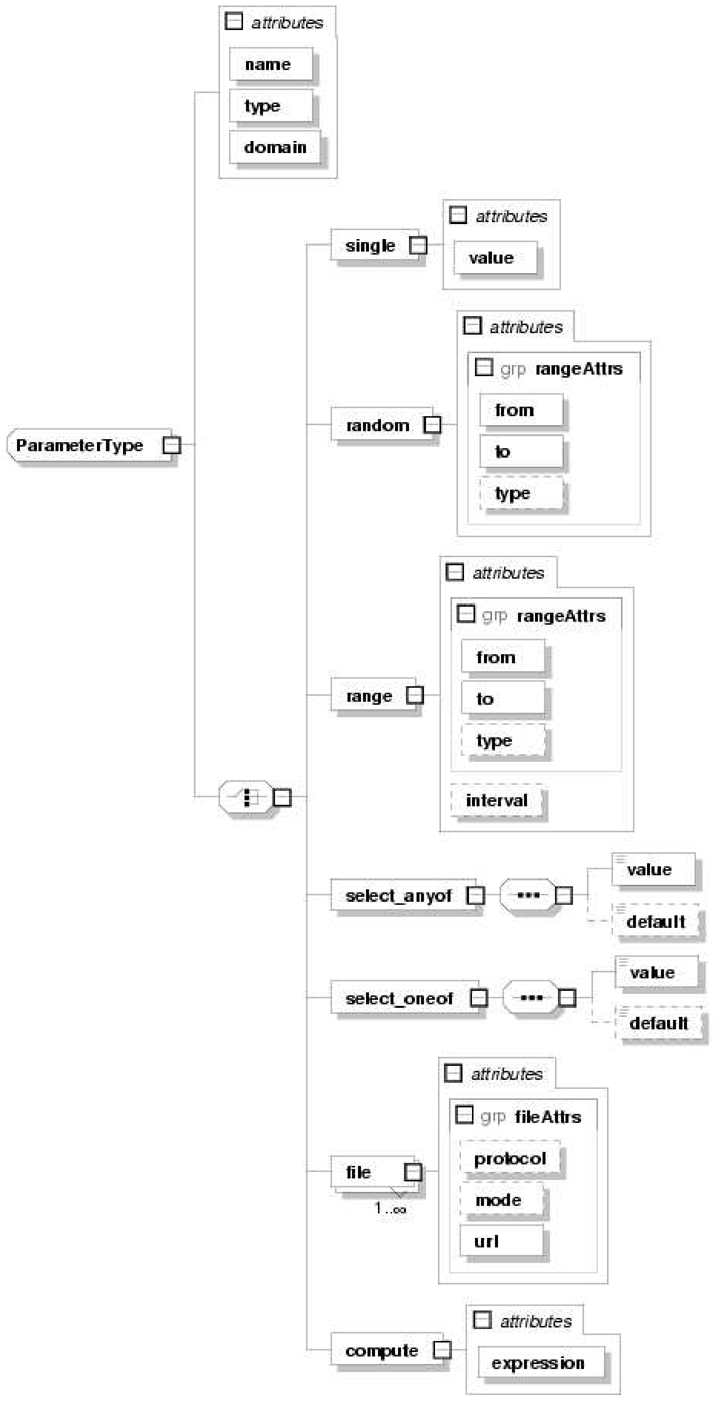


Figure 14: Broker XPML Schema Diagram Contd..

class. Currently strict interface requirements do not exist. Programmers writing interpreters are encouraged to take a look at the code of the `XMLReader` and create classes on similar lines. In future versions of the

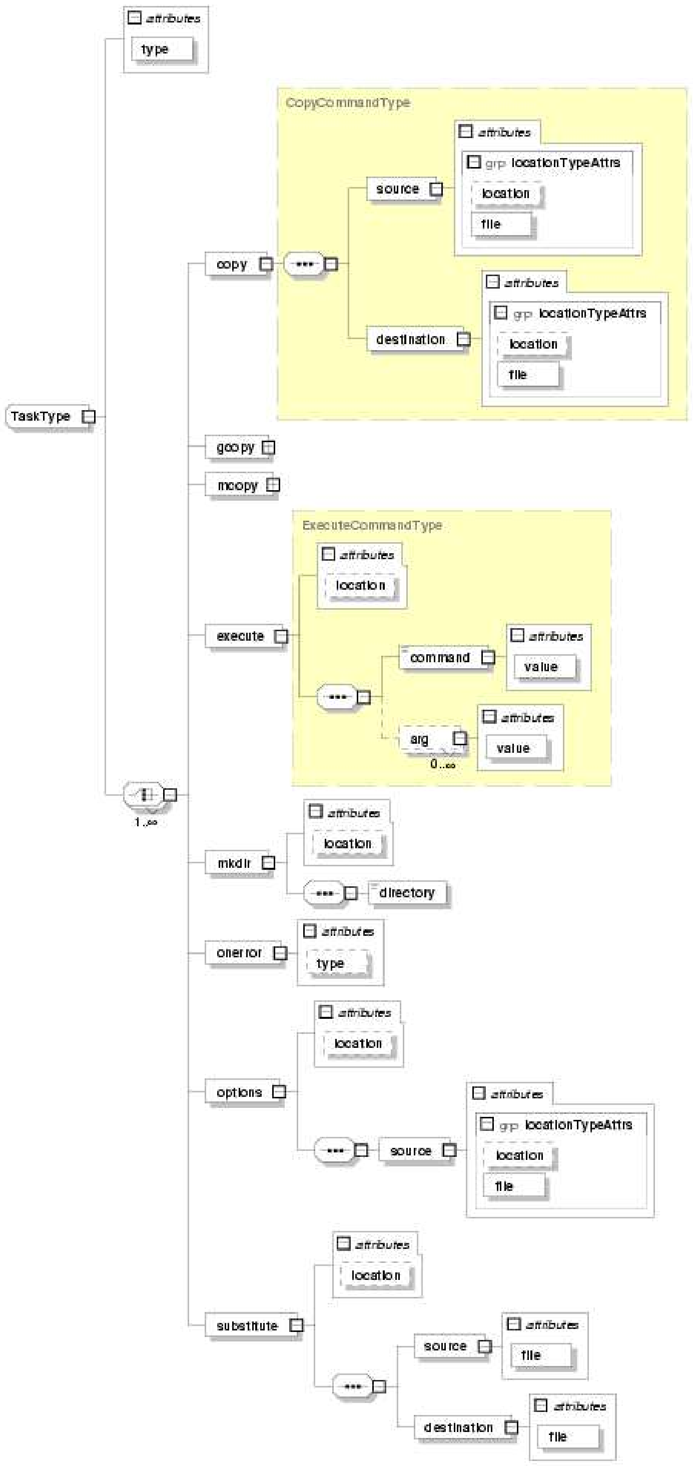


Figure 15: Broker XPML Schema Diagram Contd..

broker, it is planned to provide well-defined interfaces to write interpreters for other application descriptions.

5.4.2 Porting the broker for different low-level middleware

The current version of the broker supports Globus 2.4, Globus 3.2, Alchemi 0.8, and Unicore 4.1 (experimental support). Each of the middleware plug-ins (except Globus 3.2) has its own package in the broker; i.e the Globus related classes are in the org.gridbus.broker.farming.common.globus package, Alchemi related classes in org.gridbus.broker.farming.common.alchemi and so on. So, to support a new middleware system, one just needs to create some middleware specific classes which conform to a certain specification. This section gives directions on how to implement a new middleware plug-in and a brief description of the Globus implementation.
To support a new middleware, certain middleware specific classes need to be written for the broker. These include a middleware specific `ComputeServer` class and `JobWrapper` class. The `ComputeServer` abstract class needs to be extended and all the abstract methods are to be implemented. These include :

- discoverProperties
- queryJobStatus
- shutDownServer
- updateStatus

Each `ComputeServer` has an associated `ComputeServerManager` which is a generic middleware independent class which manages job submission to the `ComputeServer`, using a local job buffer. The `discoverProperties` method is called when the `ComputeServerManager` is started. This method is meant to set the middleware specific properties and initialize any other fields in the `ComputeServer` sub-class. This method is used to set the "alive" status of the server. Typically the server is pinged and queried for information in this method. If this method fails, it is assumed that the server cannot be contacted, and it will not be considered to be alive. Only servers which are alive are used during scheduling. The `prepareJob` method is meant to set the middleware specific `JobWrapper` for the job, and also set the server for the job. The `queryJobStatus` is used to query the status of a job executing on the remote node. It receives the job handle argument as a generic Java Object. Using the job handle, the `queryJobStatus` is meant to find out the status of the job, interpret its meaning and map it to one of the generic status codes defined in the Job class. The status codes defined are all integers and the `queryJobStatus` is expected to return an integer. The `shutDownServer` method is provided for any middleware specific clean up procedures which need to be performed once for each server before closing down the broker. The `updateStatus` method is meant for updating the status field of the `ComputeServer`. (This is provided for future use, even though the current version of the broker doesn't really use the status field. The broker mainly checks the `isAlive` method to determine the server status). In addition, some of the methods of the `ComputeServer` can be overridden if the default implementation doesn't suffice for a middleware specific server.
The other part of a middleware plug-in involves creating a class that implements the `JobWrapper` interface. The `JobWrapper` is the object that performs the actual process of sending the job to the remote node and executing it there. The methods that need to be implemented are -

- execute
- terminate

The execute method typically reads in a Job, and iterates through all its commands and variables. It then performs appropriate operations, such as creating scripts etc... to actually execute the commands in a job's task. Before executing any command on the remote node, a temporary directory should be created for each instance of the broker. The name of this directory is of the form `GB2.<brokerID>`. Each job is expected to create a sub-directory inside this broker-temp directory on the node. All the "commands" specified in the `RunFile` package are expected to be implemented in some way. Support for accessing data from remote datahosts, like SRB for example, also should be provided. The jobwrapper is expected to copy back the standard output and standard error of each job as two seperate files with names of the form : `stdout.<jobID>` and `stderr.<jobID>`. The broker's `JobMonitor` associated with each `ComputeServer` checks for the existence of these files before reporting the job's status as `DONE`. The copy back of the standard out and error are to be implemented such that they are the last things the job does on the remote node. The job itself is submitted to the remote node in "batch" mode, and a handle to the running job is returned and set via the `setJobHandle` method of the job. The execute method is also expected to initiate the monitoring of the job, set its submitted time-stamp, remote directory, and job-handle for the job:

```
job.getServer().startMonitoring(job);
job.setJobSubmittedTimestamp();
job.setRemoteDir("<set the remote dir here>");
job.setJobHandle("<some job-handle object obtained from the middleware>");
```

The execute method therefore returns immediately after sending the job to the remote node and starting execution. Since the job is expected to be submitted in batch mode, it doesn't wait till the job returns from the node. (Note: In case the middleware doesn't yet support the submission of batch jobs, and can only execute jobs interactively, the `JobWrapper` for the middleware may still try to associate some job handle with it, and return immediately after the execute method, by starting the interactive job on a seperate thread. This approach is followed in the implementation of the middleware plug-in for Alchemi v.0.8.) The terminate method is provided for any job-specific clean-up operations that may need to be done on the remote side. The terminate method is called by the `JobMonitor` after the job is reported done or failed.
As an example the Globus implementation is briefly described. The Globus implementation uses the Cog kit v.1.2 for submission and management of jobs on a Globus node. The same set of classes are used for both Globus v.2.4 and Globus v.3.2. The `GlobusComputeServer` class extends the `ComputeServer` class and implements the abstract methods `queryJobStatus, discoverProperties, shutDownServer` and `updateStatus`. It also stores the user's proxy that is used for job-submission by the `GlobusJobWrapper`.
The `queryJobStatus` method queries the job on the remote side using the Gram class of the Cog-Kit.
The following are the job-status mappings for Globus:

```
GramJob.STATUS_STAGE_IN          -  Job.SUBMITTED
GramJob.STATUS_ACTIVE            -  Job.ACTIVE
GramJob.STATUS_UNSUBMITTED       -  Job.UNSUBMITTED
GramJob.STATUS_PENDING           -  Job.PENDING
GramJob.STATUS_FAILED            -  Job.FAILED
Any other status is mapped to    -  Job.UNKNOWN
```

The `queryJobStatus` is called periodically on a job, by the `JobMonitor` once every polling interval which is determined by the broker configuration. The `GlobusComputeServer` implements the

discoverProperties by querying the MDS on the remote node. The information returned by the query is used to set the properties for the compute server. If the discover properties fails or the remote node cannot be contacted or ping-ed then the isAlive flag is set to false. The updateStatus is implemented to do nothing specific for Globus currently.
The GlobusJobWrapper implements the JobWrapper interface and performs all the steps mentioned above in the description of the implementation of the execute method. A standard *nix shell script is created which includes *nix shell commands to implement the various commands in the Runfile package. For example, copying files is achieved by using globus-url-copy on the remote node and starting up a GASS server locally (one GassServer is started for each GlobusComputeServer). SRB support is built-in by the use of SCommands which are a command line client interface to SRB. A NodeRequirements job is implemented in a similar way, except that it doesn't have its own job-directory inside the broker's temp directory on the remote node. All the jobs create symbolic links to all files in the broker temp directory (which is the parent of each job-directory on the remote node). After the shell script is created, the job RSL is generated and the job is submitted using Gram. A handle is obtained and set for the job, and the job monitor is informed to start monitoring this job. The terminate method in the GlobusJobWrapper just deletes the shell file which is stored locally. However, the shell file is not deleted if the broker is run in "debug" mode (set using the logger level to DEBUG).

5.4.3 Schedulers

The scheduler is a middleware independent component and is only concerned with allocating jobs to resources based on its internal algorithm. The design of the broker therefore allows anyone to write their own scheduler, to implement a custom algorithm. Each scheduler is a class in the org.gridbus.broker.scheduler package. The broker currently ships with three schedulers - DBScheduler, DataScheduler and the DBDataScheduler. This section describes the way to write a custom scheduler for the broker.
Every scheduler that works with the broker must inherit from the Scheduler class. The schedule method is where the algorithm is implemented. The scheduler works with the jobs and servers collections, and uses methods in the ComputeServer and Job class to find out information about a server or a job. The DataHost and DataFile classes are used in data-aware scheduling (where the scheduler takes into account the effect of moving data around the grid etc.). The implementation of the schedule method depends entirely on the algorithm chosen. The only thing that needs to be taken care of, is to check whether a server is alive or not before submitting a job to it (using the isAlive method).
The schedulers included with the broker are all "polling-based" schedulers which iterate through the entire list of jobs ad servers to select the best server that can execute a job. For each poll, the jobs which are still UNSUBMITTED are submitted to a suitable server if one is found. The submission itself doesn't depend on any middleware specific operations, as that is handled by the middleware plug-ins. The submitJob method of the ComputeServer class is used to submit a job to a server. Apart from the schedule method, the scheduler may choose to override the updateStats method to update the statistics for jobs and servers.
The farming engine selects a scheduler based on its algorithm "type". The SchedulerFactory class is used by the farming engine to get the appropriate scheduler.

5.4.4 Persistence providers

The broker's persistence feature is designed to provide scalability and fail-safe operation. The persistence model involved dumping the entire state and execution history of the running broker periodically to persistent

storage, so that the broker can be revived to its original state in case of a crash or a failure on the broker side. The broker is able to be restarted from its last known state and continue executing without much loss in terms of CPU time taken by the jobs that were being executed on the grid. The broker recovery process also takes care that finished jobs should not have to be run again. This feature becomes especially important when considering real-world scenarios and grid economy. The persistence model provided is designed to use any persistence storage provider such as a database, the local file-system etc. The Reader and Writer interfaces define a base set of APIs that need to be implemented by any persistence provider. The State class derives from the farming engine and is meant to represent the frozen state of the broker at any point of time. The default persistence provide that comes with the broker uses a relational database. The Reader and Writer interfaces are implemented by the `DBReader` and `DBWriter` classes using the generic JDBC APIS. The database-persistence feature is expected to work with any JDBC compliant RDBMS and has been tested with mySQL (v.3.22 and above). The broker database design is shown in Fig. 16:
The design has the Broker table, central to all other entities in the database. There is one record for each run of the broker in this table. The other tables store information about the main broker components : ComputeServers, Jobs, Tasks, Variables, Datahosts, Datafiles - all linked with the unique broker id. All the tables are updated periodically, by the scheduler by calling the "store" method in the farming engine.
If a database doesn't suit one's needs for persistence, it is a simple task to write a persistence provider for the broker, by simply implementing the Reader and Writer interfaces.
The recovery process involves reviving the broker to a state that was last recorded on persistent storage. The unique broker instance id is a required input for this mode of operation. The recovery follows the steps outlined below:

- read and create the broker instance (the state object) associated with the given id
- populate the state object with the configuration properties used prior to the failure
- create the servers from the saved state
- create the datahosts and datafile objects from the saved state
- create the variables, task and jobs (in that order) from the saved state
- start up the manager and monitor threads associated with each compute server
- recover the jobs which are already running on the grid. (re-query status and recover job outputs)
- resume scheduling of jobs which are not submitted.

5.4.5 Authentication mechanisms

As the broker supports various middleware, it also naturally supports the different ways of authenticating to resources, on behalf of the user. Currently the following methods of authentication with resources are implemented:

- x.509 proxy certificates generated locally
- x.509 proxies stored in MyProxy which can be retrieved by the broker
- Simple authentication using a username / password pair

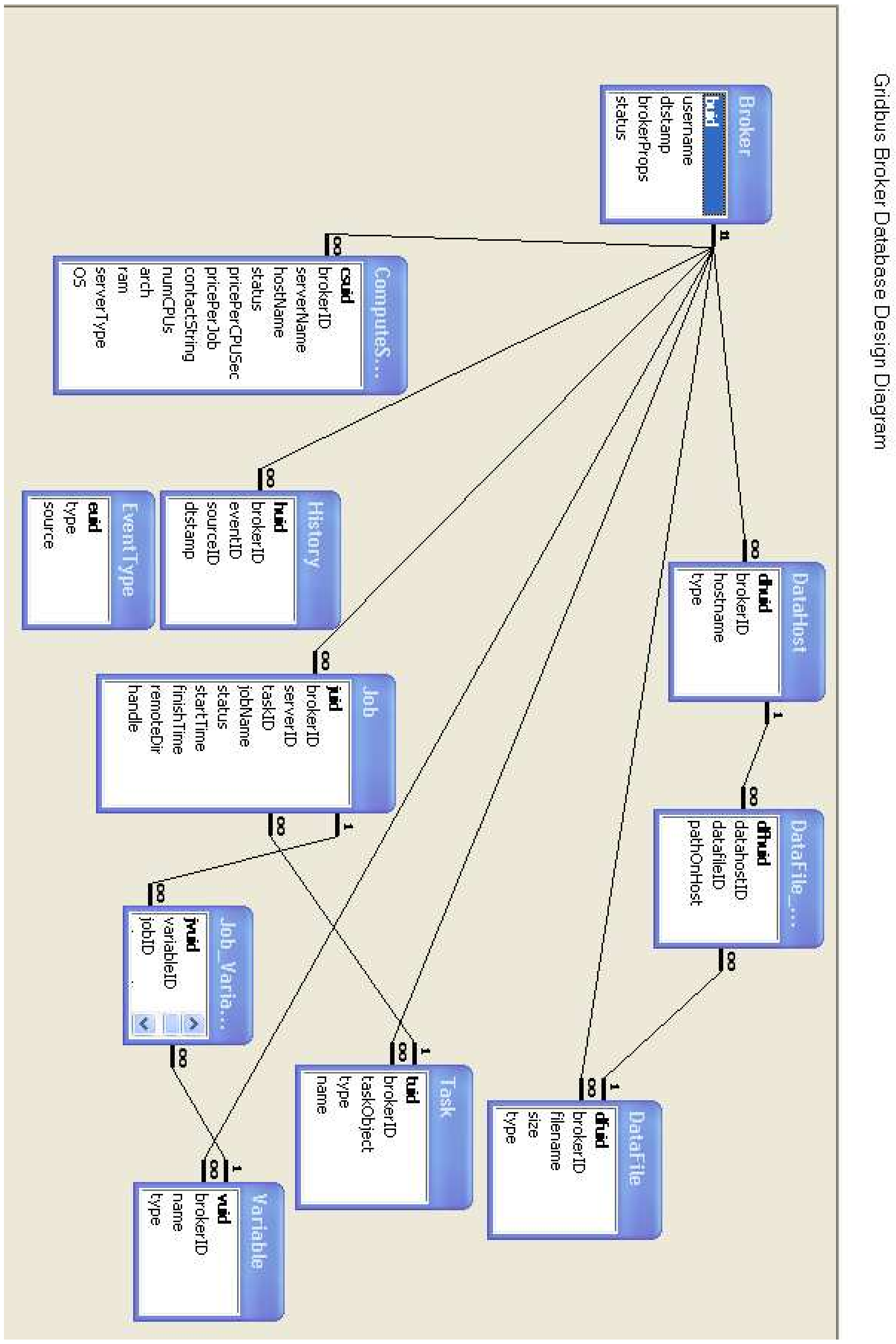


Figure 16: Broker Database Design Diagram

- Keystore-based authentication

The x.509 proxies are used for Globus resources, and GSI-enabled SRB systems. Keystores are used for Unicore nodes, while Alchemi nodes are supplied with a username/password pair for authentication.

The `UserCredential` class and its subclasses represent the various credentials that handle authentication to remote grid nodes. To be able to support other ways of authentication, the UserCredential class would need to be extended, and appropriate mechanism is to be implemented. As the broker is ported to more middleware, it will also support other authentication mechanisms as needed.

6 TROUBLESHOOTING

This section has some information on troubleshooting the broker. It will be kept up-to-date online at : `http://www.gridbus.org/broker/2.0/manual`

- If you get a ClassNotFoundException while running the broker, please make sure the broker jar and all other jar libraries are in the classpath
- When using the broker in Gridsphere portlets running on Windows, the system-user.dir is set to `C:\windows\system32` since the portlet container runs as the SYSTEM user. So, the broker temporary directory is created in the `config\systemprofile` subdirectory of the Windows system directory.

7 KNOWN ISSUES / LIMITATIONS

As with any software made by anyone, the broker has its own limitations. Some of the most obvious ones are listed here:

- The broker expects Globus certificates to be in their default directories. So, the usercert.pem and userkey.pem need to be located in the .globus directory in the user's home directory. The next version of the broker will have APIs in the `GridProxyUtil` class to specify the globus cert locations to override this default behaviour
- The grid-proxy utility in the broker uses the libraries in the Cog-Kit and Java, which echo password to the stdout. This is an inherent problem with Java and users who wish to avoid this should simply initialise the broker before starting it, so that the broker can pick up the existing proxy.
- The broker considers all paths on the local node to be relative to either its local directory or its temp directory as configured in the Broker.properties. This is mainly a problem if using the XPML to specify the application description. The "copy" commands in the XPML assume the paths are all relative to the local directory of the broker. This issue does not arise when programming the broker, since an absolute path can be set using the API.
- The Unicore implementation is not guaranteed to work, and is still shaky at best. This issue will probably take a while to resolve, since it involves setting up a Unicore node and detailed testing.
- Data-aware scheduling may not be accurate when running the broker on Windows, since the broker depends on Network Weather Service (NWS) for bandwidth data. NWS is currently not available for Windows, and hence the scheduling may not reflect the correctness of the algorithm.

- The broker can only work with one SRB catalog at present. This is going to be fixed when the XML-based resource description is implemented, in the next version.
- The full XPML schema itself is not implemented in the current version. The schema is likely to be modified and enhanced. The future versions of the broker will aim to implement most of the schema definitions.
- Globus 3.2 support will require pre-WS components on the remote node. A complete GT3.2 support using the managed-job-submission services will be offered in the next version.

8 FUTURE DEVELOPMENT PLANS

The Gridbus Broker is a project in continuous development. As part of the Gridbus project, it aims to implement the innovations that come out of the research work performed by the members of the GRIDS lab. Below is a brief about the current plans about the future development path of the broker. (The list below mainly deals with new features. Obviously there will be work going on to rid the broker of its current limitations and also improvement in existing features).

- Middleware
  - Support for Globus toolkit 4.0 and complete support for v.3.2 (using grid-service model)
  - Support for other middleware/low-level services such as Clarens, NorduGrid, CondorG, XGrid etc.
  - Implementing a middleware simulator to run applications on simulated grid nodes using GridSim
- Application Description Interfaces and Programming models
  - Extensions to XPML to include enumerated types, conditions and loops, input validation
  - Support for new application description languages and programming models such as Grid superscalar, ASSIST, GAT, BPeL4WS
  - A new Grid-thread application programming model for creating distributed applications that run on a global grid
- Data services
  - Improved support for SRB (Storage Resource Broker) datahosts including GSI-enabled SRB, optimisation of selection of SRB datahosts, and multiple SRB federations
- Programmer support
  - Cleaner and enhanced APIs for easier programmability
  - The broker APIs available as a set of loosely coupled Web-services so that programs written in any language can take advantage of the services of the Gridbus resource broker
  - A plug-in for the Eclipse development environment to facilitate programming with (and extending) the broker.
- User friendly (GUI/Web-based) tools

  - starting the broker
  - job monitoring
  - browsing and analysing the broker database and execution logs
  - generation and modification of XPML and resource description XML

- Others
  - Improved performance, scalability and robustness
  - Advanced scheduling algorithms
  - Support for Grid Marker Directory service, VO directories and other information services as Community Authorization Service(CAS).
  - Support for grid-economy model via GridBank
  - Ability to invoke any webservice using the broker dynamically at runtime
  - XML-based persistence for removing dependence on an external RDBMS

9 CONCLUSION AND ACKNOWLEDGMENTS

This manual attempts to explain the design of the Gridbus Broker, how to install, configure, and use it. The programmer's manual aims to help programmers use the broker in their own programs and how to extend the broker to tailor it to various situations. The Gridbus Broker team would be very happy to answer any queries that you may have regarding the Broker. Relevant contact information is given below.
Contact: Dr. Rajkumar Buyya(raj@cs.mu.oz.au), Srikumar Venugopal(srikumar@cs.mu.oz.au), Krishna Nadiminti(kna@cs.mu.oz.au), Hussein Gibbins (hag@cs.mu.oz.au)
We would like to take this opportunity to acknowledge all the help and support extended to us by all the members of the GRIDS lab, CSSE Department, Melbourne University. The Portlets section was contributed by Hussein on the basis of his experience of using the Broker with the BioGrid Portal development. We would to give our special thanks to Dr. Fanel Donea, Theory VO Scientific Programmer,Centre for Astrophysics and Supercomputing, Swinburne University for his comments about the broker, which has helped to make this manual better than what it would have been otherwise. We would also like to thank Dr. Lyle Winton (School of Physics, University of Melbourne) for his contribution to the intial thoughts over Broker design.
[Note: The latest version of this manual will be available online at
`http://www.gridbus.org/broker/2.0/manualv2.pdf`]

References

[1] R. Buyya, S. Date, Y. Mizuno-Matsumoto, S. Venugopal, and D. Abramson, "Neuroscience Instrumentation and Distributed Analysis of Brain Activity Data: A Case for eScience on Global Grids," *Journal of Concurrency and Computation: Practice and Experience*, 2005. [Online]. Available: http://www.gridbus.org/˜raj/papers/neurogrid-ccpe.pdf

[2] S. Venugopal, R. Buyya, and L. Winton, "A Grid Service Broker for Scheduling Distributed Data-Oriented Applications on Global Grids," in *Proceedings of the 2nd Workshop on Middleware in Grid Computing (MGC 04) : 5th ACM International Middleware Conference (Middleware 2004)*, Toronto, Canada, October 2004.

[3] B. Hughes, S. Venugopal, and R. Buyya, “Grid-based Indexing of a Newswire Corpus,” in *Proceedings of the 5th IEEE/ACM International Workshop on Grid Computing (GRID 2004)*. Pittsburgh, USA: IEEE Computer Society Press, Los Alamitos, CA, USA., Nov. 2004.

[4] B. Beeson, S. Melnikoff, S. Venugopal, and D. G. Barnes, “ A Portal for Grid-enabled Physics,” in *Proceedings of the 2005 Australasian Workshop on Grid Computing and e-Research (AusGrid 2005)*. Newcastle, NSW, Australia: Australian Computer Society, January 2005.