The Gridbus Grid Service Broker and Scheduler (v.3.0) User Guide

Krishna Nadiminti, Hussein Gibbins, Xingchen Chu, Srikumar Venugopal and 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, hag, xchu, srikumar, raj}@csse.unimelb.edu.au

http://www.gridbus.org/broker

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" services 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 services 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 services. 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 resource/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 services 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 services transparently, without having to worry about availability, access methods, security issues and other policies. The Gridbus service 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 at least one way of submitting a job and if running a flavour of Unix will provide at least a POSIX shell. Also, no assumption is made about service 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. The advantages of this are:
  - In a Grid with multiple services configured differently, the broker tries to make use of every service 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 grid node.
  - The broker can run jobs on resources with different middleware at the same time.
  - The broker needs not to be refactored if there is a new version of the middleware.

- The broker is able to handle gracefully jobs and services 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 service 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, services 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 service - 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 favourite interface to describe applications.

Figure **1** shows the block diagram of the broker. The main features of the Gridbus Broker (version 3.0) are:

- Discovery of services on the grid
- Transparent Access to computational services running middleware such as:
  - Globus 2.4
  - Globus 3.2 (pre-WS)
  - Globus 4.0
  - Alchemi 1.0
- And queuing systems such as
  - OpenPBS 2.3.
  - Sun N1 Grid Engine 6 (SGE)

  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.
- 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 services
- XML-based application description format, XML-based service description format
- Support for data sources managed by systems such as Storage Resource Broker (SRB), and the Globus Replica Catalogue.
- 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.

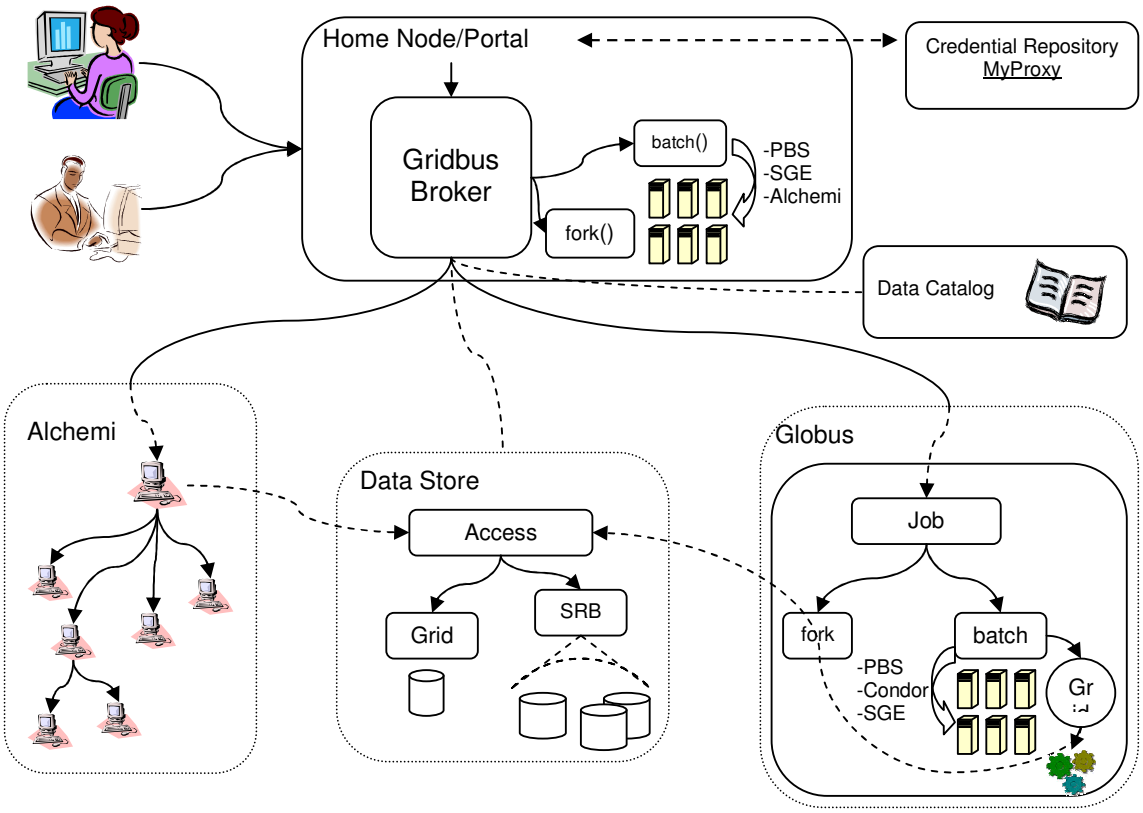


Figure 1 : Broker Block Diagram.

The Gridbus broker uses a default application-description interpreter for a language called XPML (eXtensible Parametric Modelling 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. In addition, the broker can interpret GGF-defined JSDL documents (only a subset of JSDL is supported). New application descriptions can be easily plugged-in.

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 **Figure 2**.

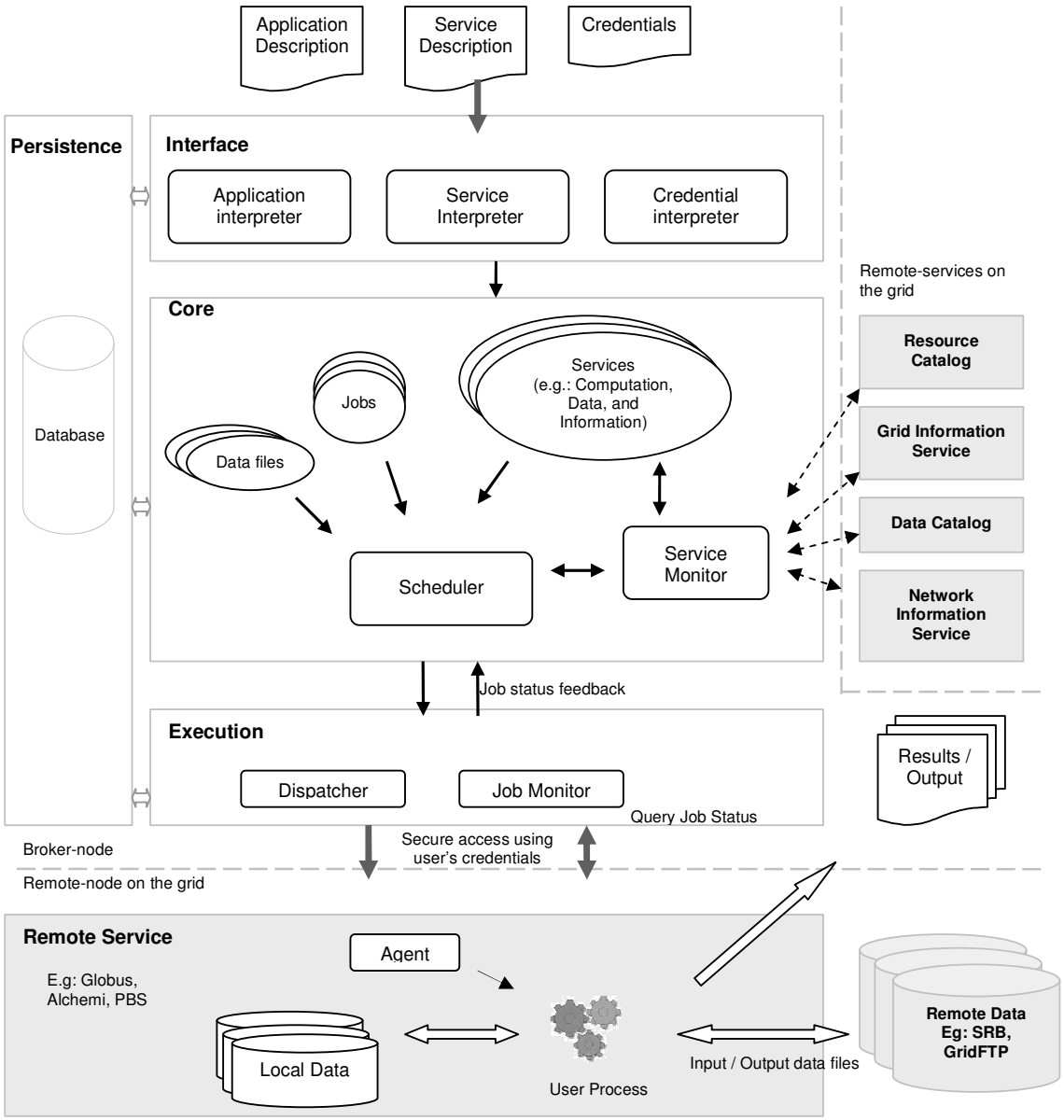


Figure 2 : Broker Architecture.

The broker can be thought of as a system composed of three main sub-systems:

- The interface sub-system
- The core-sub-system
- The execution sub-system.

The input to the broker is an application-description, which consists of tasks and the associated parameters with their values, and a resource/service description which could be in the form of a file specifying the hosts available or information service which the broker queries, and a set of credentials to use to authenticate to remote services. The interface sub-system has modules that convert these inputs into entities, called jobs and services 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 is obtained from the process of interpreting the application-description. A service represents a service on the grid, which could be a compute, storage, information or application service. The task requirements and the service information drive the discovery of services such as computational nodes, application and data

services. The service 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 credential description, whenever it needs to authenticate with a remote service. Once the jobs are prepared and the services are discovered, the scheduler is started. The scheduler maps jobs to suitable servers based on its algorithm. The dispatcher submits mapped jobs to the servers using the actuator component, in the execution sub-system, which wraps the job in an agent that is sent to the remote node. The actuator is a middleware specific component which communicates with the remote grid service using protocols understood by it. The job-monitor periodically monitors the jobs using the services of the execution sub-system and updates the broker's internal state, which is persisted to non-volatile storage when needed.

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 completed. The scheduling policy determines whether failed jobs are restarted or ignored.

1.5 Broker Usage Scenarios

The broker can operate in a number of different hardware and software configuration scenarios. Some examples include forking jobs on the local machine, or a remote machine using SSH, or submitting a job to a remote queuing system without the need of Globus. For more details on possible configurations please see APPENDIX I.

1.6 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, The University of Melbourne
- Natural Language Engineering [3] - Dept. of Computer Science, The University of Melbourne
- Astrophysics [4] - School of Physics, The University of Melbourne
- Molecular Docking (Drug Discovery) [5] - WEHI, The University of Melbourne
- Finance (Portfolio analysis) [6] - Complutense University of Madrid, Spain
- Kidney Grid (Distributed Kidney modelling) [7] - Faculty of Medicine, The University of Melbourne

It has been also been utilised in several Grid demonstrations including the 2003 IEEE/ACM Supercomputing Conference(SC 2003) HPC Challenge demonstration.

2 INSTALLATION

2.1 Requirements

Broker side (i.e. on the machine running the broker)

- Java Virtual Machine 1.4 or higher
  More info: http://www.java.com/
- Valid grid certificates properly set up (if using remote Globus nodes)
  By default the certificates are placed 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:
    - 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 for improving the accuracy of scheduling data-intensive applications that access remote data hosts)
      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 data hosts 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

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)
    - Globus 4.0
    - Alchemi 1.0 (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 (more info: http://www.cs.wisc.edu/condor/downloads/)
    - Open PBS 2.3 (more info: http://www.openpbs.org/)
    - SGE Sun N1 Grid Engine 6
      (more info: http://www.sun.com/software/gridware/index.xml)
    - XGrid Technical Preview 2 (experimental support for 1.0)

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. The installation just involves unzipping the files to any directory and optionally setting the `PATH` environment variable to include the broker executable script (gbb.sh or gbb.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-3.0.tar.gz
```

- The following directory structure is created under the main gridbusbroker-3.0 directory:

```
<install-dir>
      /bin        Scripts to run the broker, setup the database etc.
      /docs       API docs
      /examples   Examples of xml description, config files, api etc.
      /lib        Broker jar and its dependencies
      /licenses   Third party licenses
      /manual     Broker User and programmer's manual
      /src        Broker source code
      /xml        XML schemas for the broker's application, service and
                  credential description files.
```

- Set the `GBB_HOME` variable to the directory where you have installed the broker.
- Additionally, it is recommended to have the directory `gridbus-broker2.4/bin` added to the system `PATH` variable.

For example, for a Bash shell:

```
$ export PATH=$PATH:<broker-install-directory>/bin
```

- Set the permissions for the all the scripts:

```
$ chmod 755 bin/*.sh bin/util/*.sh
```

2.3 Recompiling the broker with ant

The broker is distributed with the java source code. An Ant build file is provided, so running "ant" will recompile the broker jar, and copy it to the "dist" directory. The class files built are copied to the "build" directory. There are separate build files for the broker and the WSRF broker service which can be used to build just the broker or both the broker and the WSRF service.

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 web portals. This section describes the use of the Gridbus Broker in command-line and embedded (within a Java app) modes.

The first thing that needs to be done, is to setup the broker database. The Broker uses an embedded HSQL server, which is easy to setup : from the broker-install-directory, run:

```
$ bin/util/setupDB.sh
```

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 class path 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 (found in the 'lib' directory). 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 XML 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/apps/calc.xml
```

- The Service Description:

The Service description specifies the available services on the grid, and is provided to the broker in an XML format. A service description file has a description of the services that are to be used by the broker for executing the grid application. The broker distribution has a service description template which are has to be modified to specify the services the user has access to. For example:

```
examples/services/services.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>\$ gbb.sh -a examples/apps/calc.xml
-s examples/services/services.xml -c examples/credentials/credentials.xml
```

For Windows:

```
C:\<broker-install-dir> gbb.bat -a examples\apps\calc.xml
-s examples\services\services.xml -c examples\credentials\credentials.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 The Broker Workbench (a Java GUI for the Broker)

The broker distribution comes with a GUI called Gridbus Broker Workbench which is a convenient MDI java swing GUI for users to prepare, execute and monitor a new grid application using the broker. There are some useful features that the workbench provides to the end user:

- A visual XML editor to create and modify the description files (XPML, XGRL, XCL, JSDL and ADL) required by the broker.
- A broker execution manager console that can execute and monitor the grid application running on top of the broker.
- A set of dynamic visual charts to demonstrate the statistics information of the application execution.

To run the workbench, the user can just utilize the shell script provided in the distribution with “–g” argument.

Visual XML Editor

Once the user launches the broker workbench, user will see the main window. Users can now open the XML editor to prepare, modify and validate their application (XPML, JSDL and ADL), services (XGRL) and credentials (XCL) description files. Users can have difference views against the xml file including the tree, source, structure and graph view. Fig.1 shows a example of editing a XPML file. Users can add element or attribute to each named node (for example, the task node), the editor guarantees that only the valid sub elements or attributes appear while adding them. As soon as the user have completed, all the xml files can be validated before saving them on to the disk or sending to the broker.

Broker Execution Manager Console

Besides editing the description files, the workbench also allows users to execute and monitor their newly created grid application. Within the console, users can get the basic information related to Qos, Jobs and Servers. Furthermore, as long as the users click the job or the server name, another two windows will show more details about the job and server. In addition, the application logger area is also refreshed and scrolled automatically during some period of time for user to know the exactly information of the execution of the broker. To use the facilities related to the application execution and monitor, uses just need to choose the items in “application” menu.

Statistics Charts

Another interesting feature that the workbench provides is the dynamic visual charts for the statistics information related to the application execution such as job status, job completion rate for server, and Qos (budget and schedule). The screenshots above show the job status and job running on server charts. Users can choose other charts under the “statistics” menu.

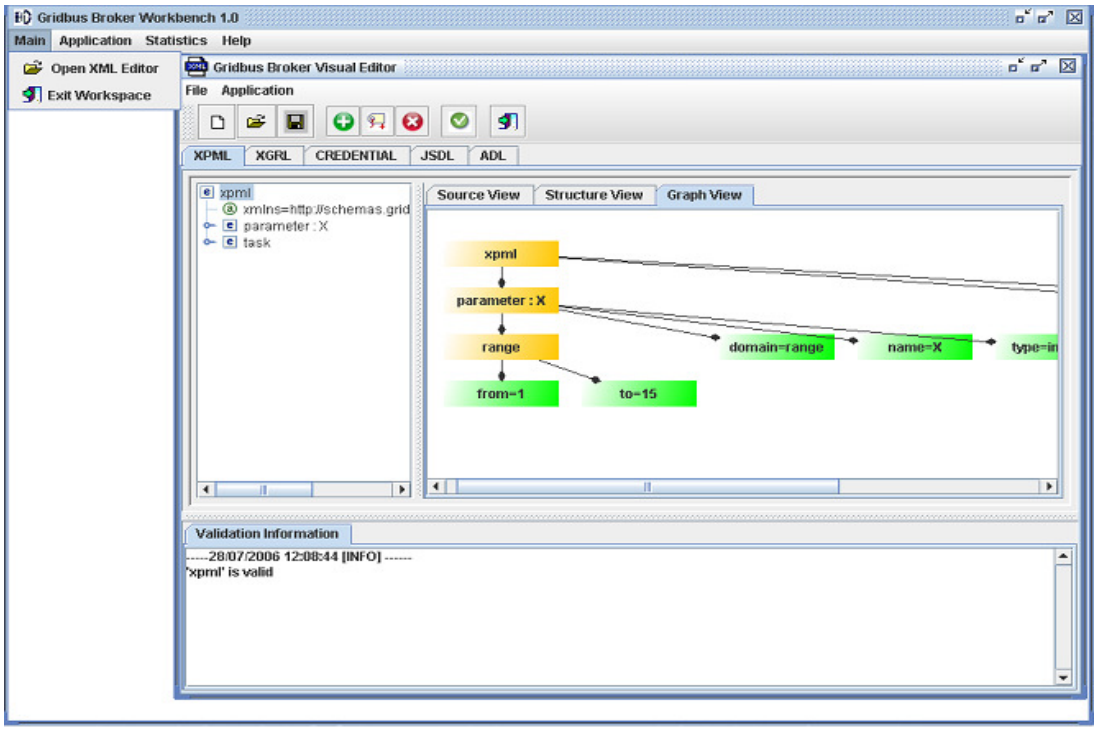


Figure 3: Visual XML Editor.

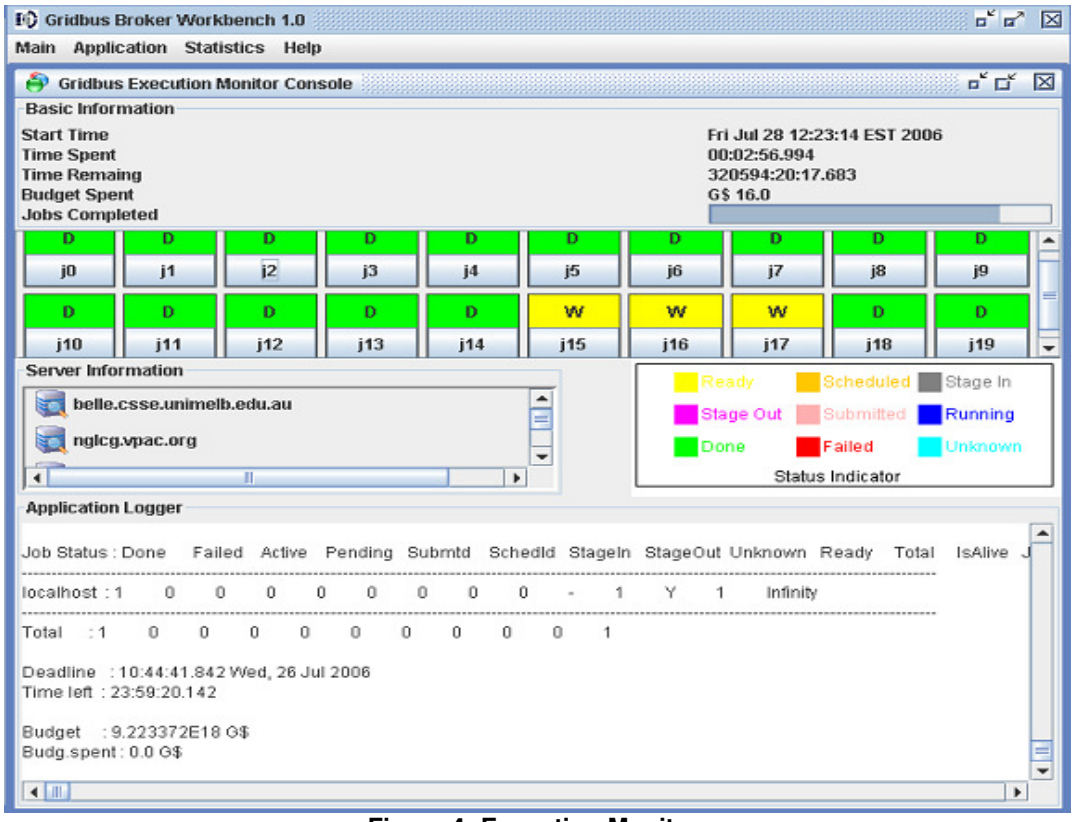


Figure 4: Execution Monitor.

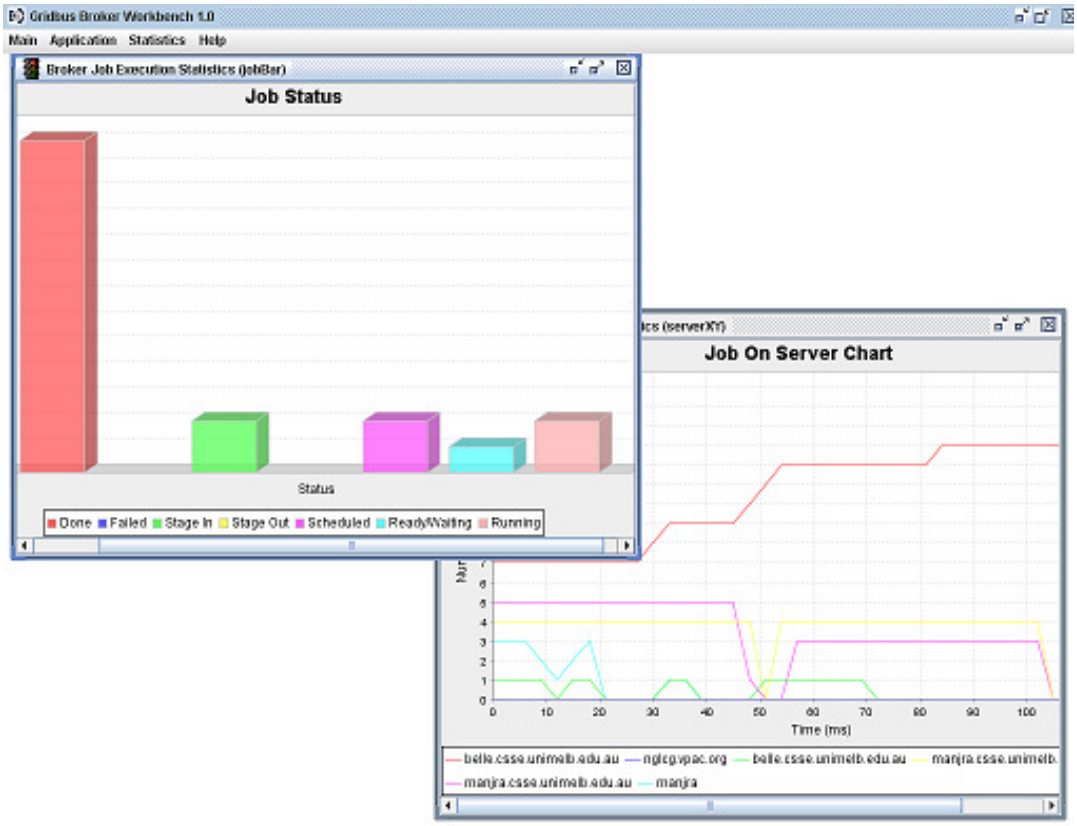


Figure 5: Performance Statistics.

3.3 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. The full functionality of the XPML files is 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 `gridbusbroker.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" for a 'single' application Broker
       //For a multi-application Broker, use the Broker class.
       GridbusFarmingEngine fe =
              GridbusFarmingEngine.createInstance(
                     "calc.xml",
                     "services.xml",
                     "credentials.xml");

       System.out.println("Starting broker scheduling...");

       //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()){
          Thread.sleep(5000);
      }

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 web application. The programmer's manual has a more detailed explanation of how to use the common broker APIs. Those who want to use the APIs are suggested to first read through the user manual and then go on to the programmers manual which has more detailed explanation of the broker architecture and common APIs. The last section of the programmer's manual also has descriptions of how to extend the broker's APIs and build upon them to suit your needs.

3.4 Remotely hosted Broker WSRF Service

Starting from version 3.0, the Gridbus Broker can be hosted as a remote WSRF service. This allows applications to avoid managing the dependencies and configuration on the client machine, and using the Broker web service. The Broker WSRF Service has been developed using the Globus WSRF implementation and tooling and can be hosted in any WSRF container. To build and deploy the Broker WSRF Service, you will need the WSRF Core Java implementation. The *wsrf-broker-service-build.xml* distributed with the broker assists in building and deploying the broker. The broker-specific configuration for the WSRF Service is similar to the case when the broker is run on the command-line. The broker command-line starter, can also be used as a WSRF client (see section 4.1). This makes use of the WSRFClient API that is found in the distribution.

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:

```
gbb [-g] [-a <application xml file>] [-r <appID>] [-s <services xml file>] [-c
<credentials xml file>] [-u <broker-service-url> [-e <epr-file> ] ]

-a,--application <arg>          Application description file in XPML format
                                (optional).
                                Default: application.xml

-c,--credentials <arg>          Credential description file in XML (optional).
                                Default: credentials.xml

-g,--gui                        Runs the broker in a graphical interface
                                (optional).
                                Default: <no-value>. Broker runs in command-line mode.

-h,--help                       Prints the usage / help and exits.

-r,--recovery <arg>             Activates Recovery mode with given id     (optional).
                                Default: normal mode, i.e starts a new    application

-s,--services <arg>             Service description file in XGRL (optional).
                                Default: services.xml

-v,--version                    Prints the current Gridbus broker version and
                                exits.

-u, --url <arg>                 Runs the broker WSRF client when the Broker WSRF
                                Service is hosted at the given URL.
                                Default: <no-value>. The broker is run locally

-e, --epr <arg>                 When combined with the -u option, specifies the
                                'application' to connect to on the remotely hosted
                                Broker WSRF Service.
                                Default: <no-value>. The broker is run locally.
```

4.2 The Broker input, output and configuration files

The main 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 Service description file format
- Configuration files (**Note**: the default ones distributed should work fine for most cases)

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

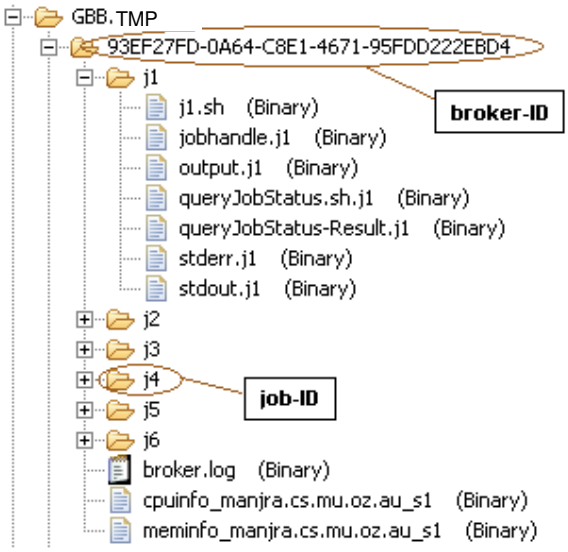
The output files produced by the broker include:

- The execution logs from the broker itself (broker.log)
- The standard output and standard error files for each job that runs on the grid
- The script files generated by the broker for each job (these files exist only if the broker is run in DEBUG mode. Otherwise they are deleted).
- Any other output files produced by the application that runs on the grid by the broker

Each run of the broker creates a separate directory under a temporary directory, which is in the current directory. This directory has a structure as follows:

```
GBB.TMP/
        <broker-ID>/
                <job-ID>/
                        stdout.<job-ID>
                        stderr.<job-ID>
                .
                .
                .
broker.log
<other-temporary files>
```

For example:



where
brokerID is *93EF27FD-0A64-C8E1-4671-95FDD222EBD4*
jobID is *j1, j2 … j6*

All the output files for each job are collected in its own directory, ensuring clean separation of files.

[Note: The SAME directory structure (with a prefix "REMOTE.") is also created on the remote side, to achieve separation of files from different jobs and applications.]

4.2.1 The Broker configuration files

The broker needs the following configuration files:

- Broker.properties
- Broker.log4j.properties
- Broker.hibernate.cfg.xml (for configuring hibernate persistence)

The broker distribution comes with a default set of these files which will work in almost all cases. Logging for the broker is provided by the log4j library and persistence by the hibernate system. The log4j properties and hibernate configuration and object mapping requires detailed knowledge of the log4j and hibernate systems. The description of these files is out of the scope of the user manual. The programmer's manual has some discussion of the hibernate mappings the broker uses. This section briefly describes the Broker.properties file.

The Broker.properties file is 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 in figure 3. In the Broker.properties config file shown above, the broker is configured to look for the application-description file named `calc.xml` in the `examples/apps` directory relative to the current directory from where the broker is executing. The service description option points to the `services.xml` in the `examples/services` directory in this case. 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.

[**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.]

```
####
## Polling interval (optional)
## - the time interval in milliseconds for scheduler polling.
Reducing this
##   value will cause the broker to query job status' more frequently
but will
##   result in more I/O and processing.  Find a balance suitable to
your application.
####
SCHEDULER_POLL_INTERVAL=5000

####
## The PERSISTENCE_DB option specifies whether an in-built database
should be
## started by the broker (AUTO), or the user will use an existing db
server (MANUAL)
## The Default value is AUTO. Users requiring good performance should
consider
## using a seperate RDBMS server such as MySQL etc. Appropriate
changes will need to be made in the
## Broker.hibernate.cfg.xml file to specify database driver
connection settings.
## If this setting is omitted, the default setting (AUTO) is used.
PERSISTENCE_DB=AUTO
```

Figure 6: 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 parametric applications. Simply put, an XPML application description file is an XML file with special elements as defined in the XML Schema 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:

An XPML app-description consists of various sections such as:

"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 and SRB.

[Note: The name of the child element must match with the value of the domain attribute of the parameter element.]

```
<?xml version="1.0" encoding="UTF-8"?>
<xpml xmlns:xsi="http://www.w3.org/2001/XMLSchema-instance"
xmlns="http://schemas.gridbus.org/xpml/2006/01/xpml"
xsi:schemaLocation="http://schemas.gridbus.org/xpml/2006/01/xpml
XPMLSchema.xsd">
<parameter name="X" type="integer" domain="range">
      <range from="1" to="15" interval="1"/>
</parameter>
<parameter name="time_base_value" type="integer" domain="single">
      <single value="0"/>
</parameter>
<task>
      <substitute>
            <source file="output"/>
            <destination file="output.$jobname"/>
      </substitute>
      <copy>
            <source location="local" file="calc.$OS"/>
            <destination location="node" file="calc.$OS"/>
      </copy>
      <execute>
            <command value="./calc.$OS"/>
            <arg value="$X"/>
            <arg value="$time_base_value"/>
      </execute>
      <copy>
            <source location="node" file="output"/>
            <destination location="local" file="output.$jobname"/>
      </copy>
</task>
</xpml>
```

A grid application can have 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 15. The second parameter has a constant value "0". So, the number of jobs created is 15 x 1 = 15 jobs. (the first parameter can take 15 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" url="srb:/db*.jar" >
  </parameter>
```

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

```
  <parameter name="infile" type="gridfile" domain="file">
    <file protocol="srb" url="srb:/db*.jar"/>
    <file protocol="lfn"
url="lfn:/somedirectory/someotherdirectory/abc*.txt"/>
    <file protocol="srb" 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, execute, substitute etc.

- A "copy" command specifies a copy operation to be performed. Each of the copy commands has a source and destination file specified.
- 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 "$" pre-fixed. Apart from this, certain special default parameters/variables are also defined, such as :
    - `$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, and 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.

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>
        <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 used to specify the resource filtering requirements for a job. This is specified as a set of name=value pairs as shown below:

```
    <job-requirements>
        <property name=”minmemory” value=”256” />
        <property name=”maxmemory” value=”1024” />
    </job-requirements>
```

[**Note**: Currently the job-requirements feature works with only Globus. The supported parameters are the same as the ones supported in Globus RSL.]

4.2.3 The Service description file

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

- Compute services
- Data services
- Information services
- Application services

A sample service description is shown below.

```
<?xml version="1.0" encoding="UTF-8"?>
<ServiceDescription xmlns="http://schemas.gridbus.org/xgrl/2006/01/xgrl"
xmlns:xsi="http://www.w3.org/2001/XMLSchema-instance"
xsi:schemaLocation="http://schemas.gridbus.org/xgrl/2006/01/xgrl
ServiceDescriptionSchema.xsd">

      <service type="compute">
            <compute middleware="globus">
                  <globus hostname="belle.cs.mu.oz.au"
fileStagingURL="gsiftp://belle.cs.mu.oz.au//tmp" />
            </compute>
      </service>

      <service type="compute">
        <compute middleware="globus">
                  <globus hostname="manjra.cs.mu.oz.au" version="4.0"

      fileStagingURL="gsiftp://manjra.cs.mu.oz.au//tmp"
                        jobmanager="jobmanager-pbs">
                        <queue name="workq" cost="1" priority="1"
limit="50"/>
                  </globus>
        </compute>
    </service>

</ServiceDescription>
```

[**Note**: One can observe here, that the value of an attribute of a parent element generally determines which child element can be placed inside it. For example, the "type" attribute of a "service" element, determines whether a "compute", "storage", "information" or "application" child element can appear within a compute element. This pattern is followed throughout the service description schema. Credentials to access these services are specified in a separate file. This separation of credential and service elements helps to support situations where the same credential, such as a proxy certificate or a username/password pair, is to be used for authenticating to more than one service. It is common experience that such a situation is frequently seen.]

Compute services are servers to which the users' jobs can be submitted for execution. Storage services are used to store the results of execution, and hence can be considered as data sinks. Information and application services are those which provide generic services that can be used by the broker.

A “compute” service is associated with a “middleware” where the name of the middleware is to be specified. In the above example it is globus.

To describe a globus node a description similar to the following, is used:

```
<service type="compute">
        <compute middleware="globus">
                    <globus hostname="manjra.cs.mu.oz.au" version="4.0"
                           fileStagingURL="gsiftp://manjra.cs.mu.oz.au//tmp"
                           jobmanager="jobmanager-pbs">
                           <queue name="workq" cost="1" priority="1"
limit="50"/>
                    </globus>
        </compute>
    </service>
```

In the above service description, the node manjra.cs.mu.oz.au is specified to be running globus v.4.0. Similarly other compute services can be described as defined in the schema. The “type” attribute can be any of “compute”, “information” and “application”. The middleware type tag can optionally contain a set of queues. The “jobmanager” attribute of this tag identifies the queuing system. If no queue tags are defined, the available queues on that node will be discovered at run time. Otherwise, the queues can be defined explicitly as shown below, providing a higher level of control.

[**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 service description. Hence, the need for explicitly specifying data sources in the service description is obviated.]

Information services are typically entities which provide information about other 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.

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

5 TROUBLESHOOTING

This section has some information on troubleshooting the broker. Please refer to the users lists on source-forge for more up-to-date information on the broker.

- If you get a “`ClassNotFoundException`” while running the broker, please make sure the broker jar and all other jar libraries are in the classpath
- If you get a “`source: not found`” error, it means you are not running the “bash” shell. ‘source’ is a bash command which sources the commands in a shell script. In the broker, “source ” is used to set the appropriate classpath for the broker to find all the jars and libraries. In such case, please change the existing shell to “bash” shell.
- If you get an error like-

```
Cannot find log4j.properties. Logging will not be setup. Continuing...
 Creating configuration from hibernate.cfg.xml...

Exception in thread "main" org.hibernate.HibernateException: could not find
file: hibernate.cfg.xml
```

It means the database config file is not found and the classpath is not set properly. Since the config file is located in the classpath. You need to set the GBB_HOME environment variable to the directory in which you installed the broker. For example, if all the broker directories and files are in /home/belle/gridbusbroker-3.0, the GBB_HOME is /home/belle/gridbusbroker-3.0.

- While running the broker, if you get an "`Error creating storage instance`" message, it means the database is not set up properly. Normally you should run the setup script from the broker's home directory. Because the database is created where-ever you actually run the setup script! That means if you run the setup script from a certain directory, then the application should also be run from the same directory.

6 KNOWN ISSUES / LIMITATIONS

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

- The Condor, Unicore and XGrid implementations are NOT supported in this version. Support for these middleware will be evaluated as time goes on, and updated only if someone from the user-base is willing to maintain it.

- Under a Windows environment, data-aware scheduling may not be highly optimised, 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.

7 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).

- Application Description Interfaces and Programming models
    - Support for new application description languages and programming models such as Grid superscalar.

- Others
    - 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

8 CONCLUSION AND ACKNOWLEDGMENTS

This user manual attempts to explain the main concepts in the Gridbus Broker, including how to install, configure, and use it. The Gridbus Broker team would be very happy to answer any queries that you may have regarding the Broker. For more details please contact:
Dr. Rajkumar Buyya (raj), Srikumar Venugopal(srikumar), Krishna Nadiminti (kna), Hussein Gibbins (hag), and Xingchen Chu (xchu).

(append @csse.unimelb.edu.au for emails).

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. We would like to thank Dr Lyle Winton, School of Physics, and Brett Beeson from QUT for their comments and contribution.

9 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.

[5] H. Gibbins, K. Nadiminti, B. Beeson, R. Chhabra, B. Smith, and R. Buyya, The Australian BioGrid Portal: Empowering the Molecular Docking Research Community, Technical Report, GRIDS-TR-2005-9, Grid Computing and Distributed Systems Laboratory, University of Melbourne, Australia, June 13, 2005.

[6] Rafael Moreno-Vozmediano, Krishna Nadiminti, Srikumar Venugopal, Ana B Alonso-Conde, Hussein Gibbins, and Rajkumar Buyya, Distributed Portfolio and Investment Risk Analysis on Global Grids, Technical Report, GRIDS-TR-2005-14, Grid Computing and Distributed Systems Laboratory, The University of Melbourne, Australia, Nov. 14, 2005.

[7] Xingchen Chu, Andrew Lonie, Peter Harris, S.Randall Thomas, Rajkumar Buyya, KidneyGrid: A Grid Platform for Integration of Distributed Kidney Models and Resources, Technical Report, GRIDS-TR-2006-13, Grid Computing and Distributed Systems Laboratory, The University of Melbourne, Australia, Aug. 22, 2006.

APPENDIX I

The following table identifies a number of possible hardware/software configuration and provides details on how the Gridbus Broker can be utilised within such configurations and what steps need to be taken. Although Gridbus Broker is primarily designed to support global Grids, it can also be used to schedule applications on local nodes or one or more remote clusters. This ensures scalability of broker from the user desktops to global Grids.

OS Platform		**Hardware Configuration**			**Gridbus Broker Solution**
Win32	*Nix	Single server	Batch cluster	LAN-based distributed system	
■	■	■	■	■	If you have multiple systems whose configurations may be varied, the broker can utilise these different systems at the same time. Figure 7shows the broker interacting with three different hardware configurations and also different OS platforms. Individual configurations are also described in this table.
	■		■		▪ Install OpenPBS (http://www.webmo.net/support/pbs.html) or PBS Pro (http://www.openpbs.org/about_pbspro.html) or Sun N1 Grid Engine (http://www.sun.com/software/gridware/index.xml) on the batch cluster, and enable the shared file system. For the broker, use *PBS* or *SGE* resources (refer to this manual). If the broker is not installed on the head node of the cluster, configure the head node to enable SSH (http://www.openssh.com/) connections from outside. Then configure the broker to use an *SSH* dispatcher instead of a local dispatcher (which is the default setting). ▪ If Globus Toolkit 2.4.x (http://www.globus.org/toolkit/downloads/2.4.3/) has been installed on the batch cluster, simply use *GLOBUS_2_4* resource in the broker (refer to this manual). ▪ If Globus Toolkit 4.x (http://www.globus.org/toolkit/downloads/4.0.2/) has been installed on the batch cluster, simply use *GLOBUS_4_0* resource in the broker (refer to this manual). For running the broker from a client of any of the above middleware see Figure 10 and Figure 8. Executing from a non-client will first require an SSH to a client node as shown in Figure 9.
	■	■			Use the *Fork* resource in the broker to submit and execute jobs on the local machine Figure 11. If the broker is not installed on the server, configure the server to enable SSH (http://www.openssh.com) connections. Then configure the broker to use an *SSH* dispatcher instead of a local dispatcher

					(which is the default setting) Figure 12.
■				■	Install Alchemi 1.0 (http://www.alchemi.net) Manager on the selected head node; install Alchemi 1.0 Executor on the other nodes and make sure they've connected to the manager. For the broker, use *Alchemi* resource which will interact with the manager via web services or the command line client (refer to this manual). See Figure 13, Figure 14 below.

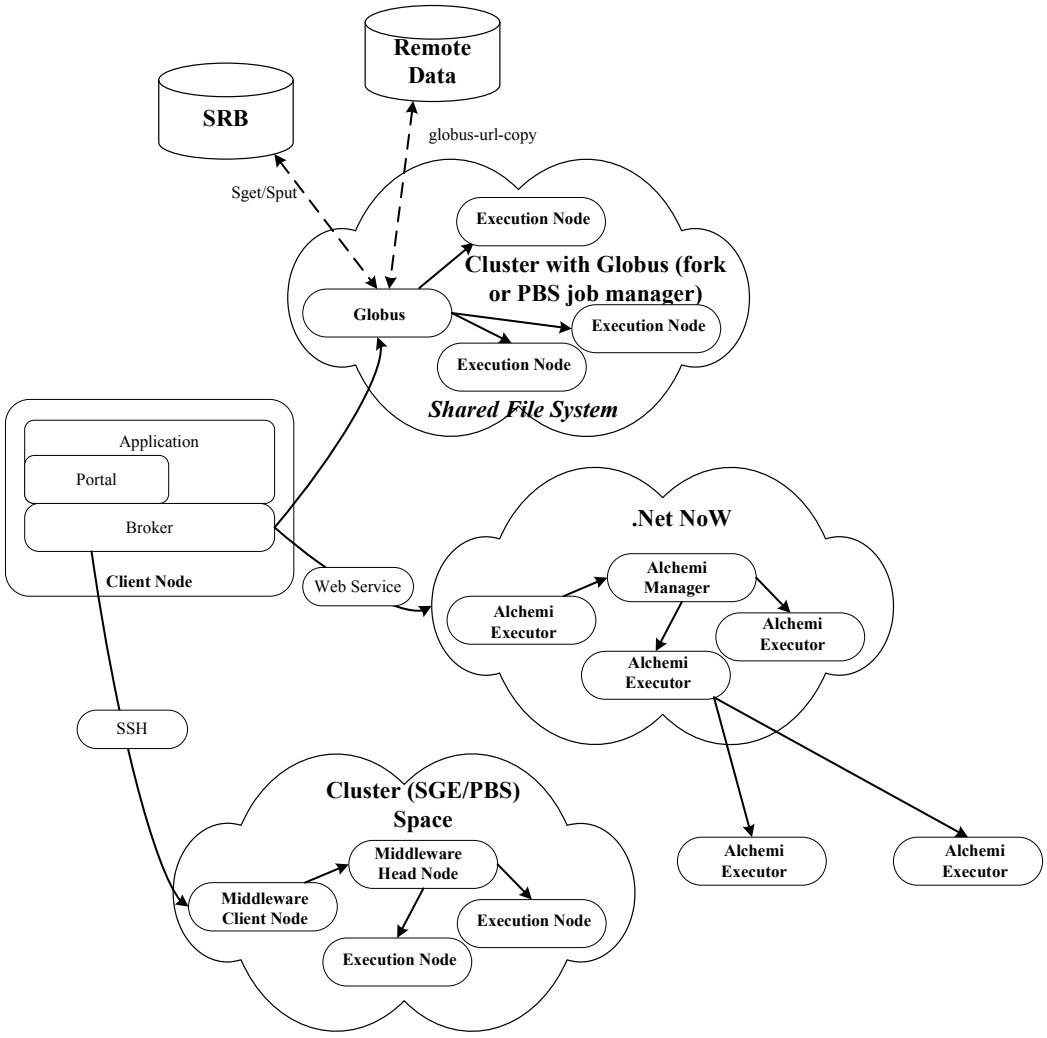


Figure 7: *Global Grid* - Utilising multiple hardware/software configurations at the same time.

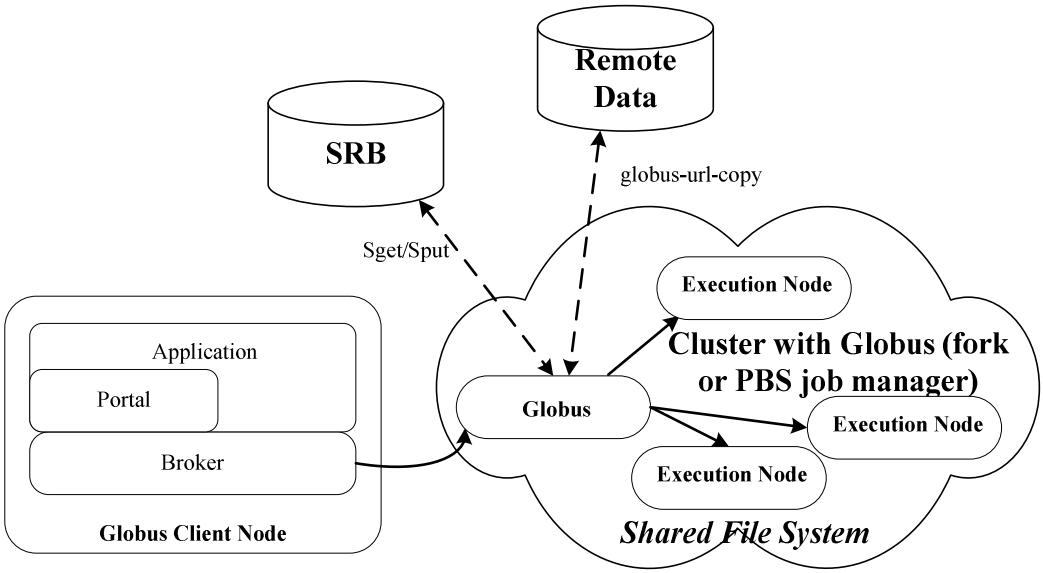


Figure 8 : Running the broker from a Globus client.

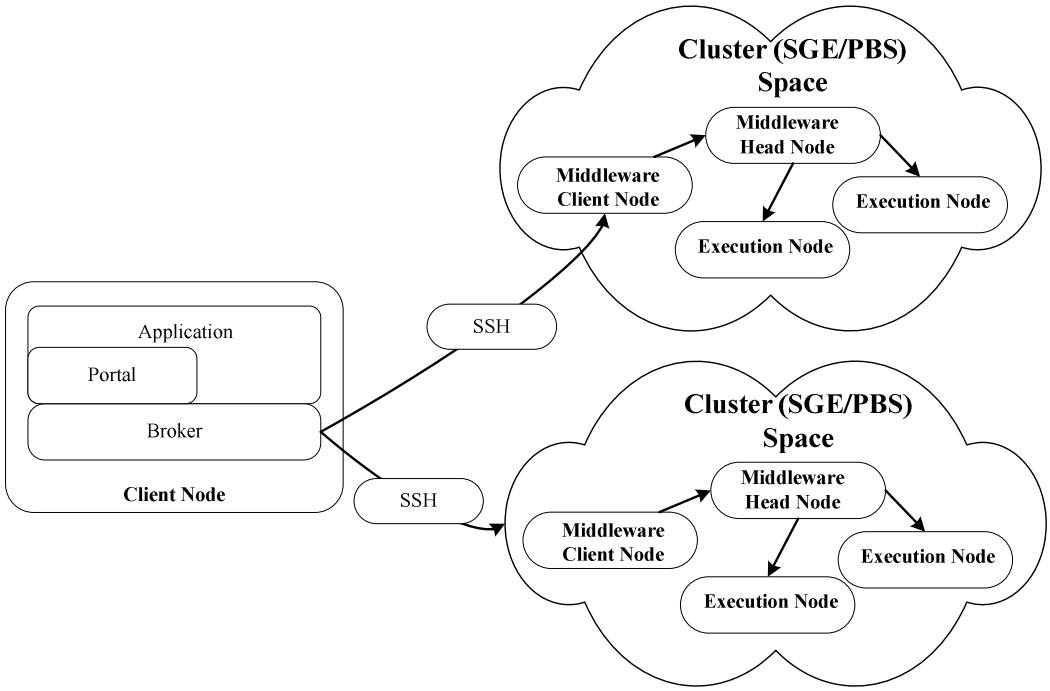


Figure 9: *Multi-clustering* - Running the broker from a non-client of the target middleware.

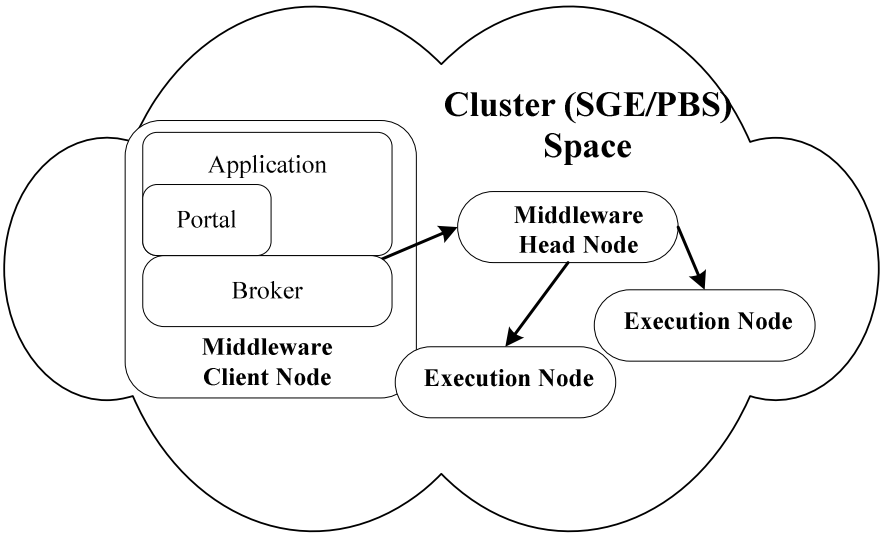


Figure 10: Running the broker from a client of the target middleware.

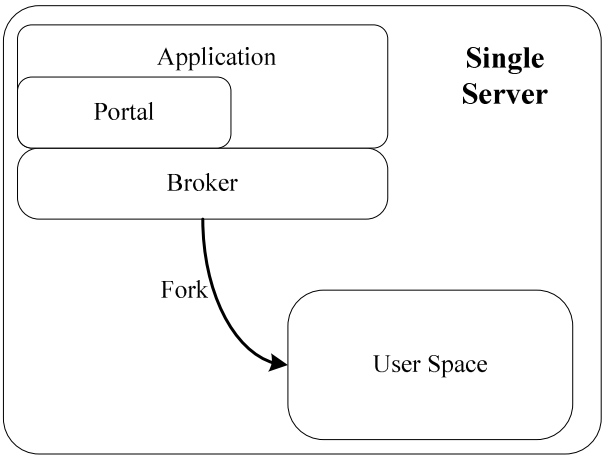


Figure 11: Forking jobs on the Local machine (*nix only).

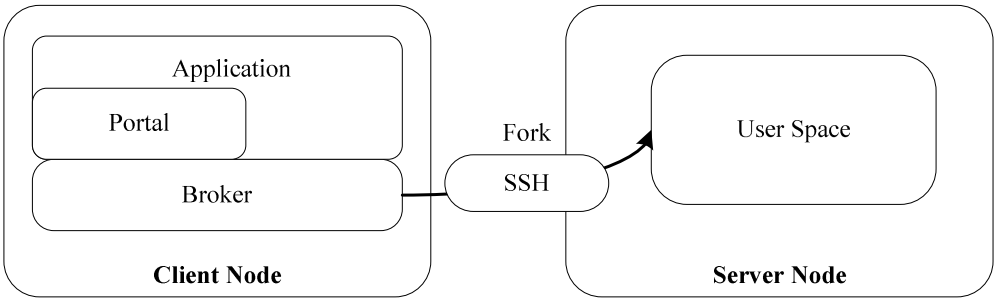


Figure 12 : Forking jobs on a remote machine using SSH.

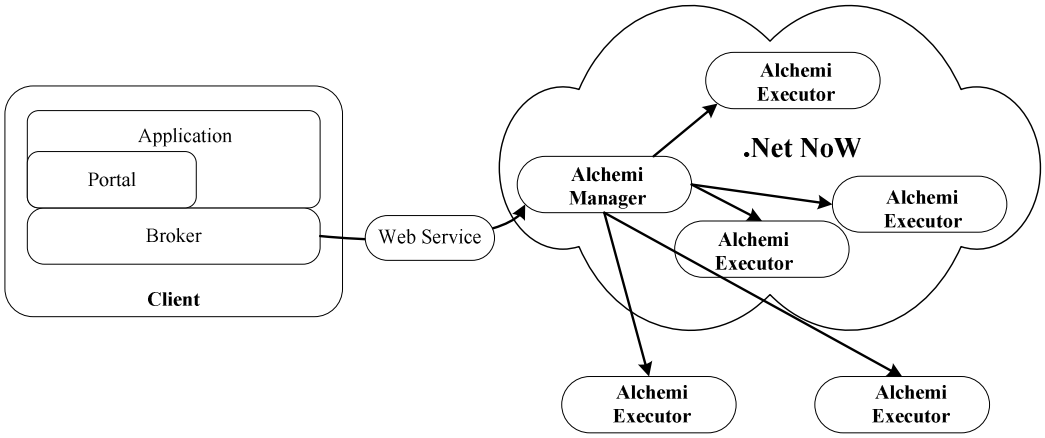


Figure 13 : Running the broker to submit to Alchemi middleware via Web Services.

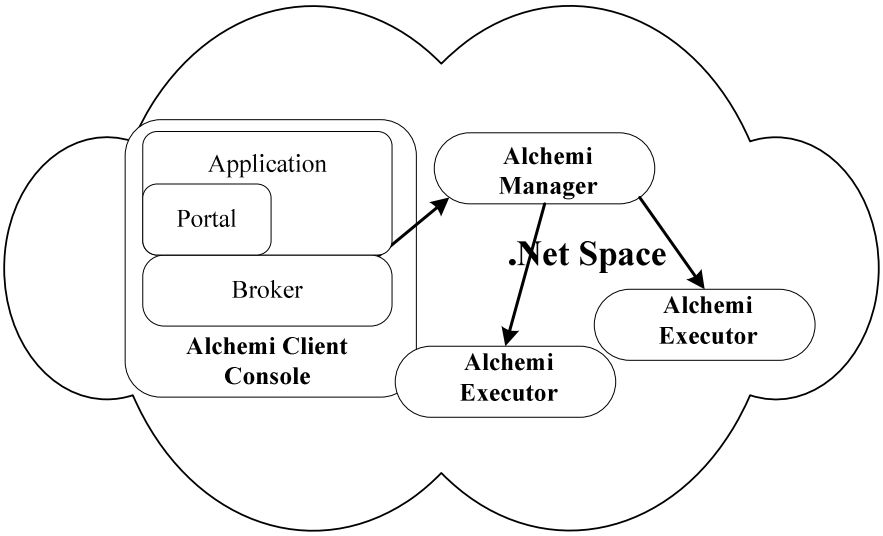


Figure 14 : Running the broker to submit to Alchemi using command line client.