Resource Broker Architecture and APIs

S. Cavalieri and S. Monforte
University of Catania – Faculty of Engineering
Department of Computer Science and Telecommunications Engineering (DIIT)

Wednesday, 13 June 2001

1. Introduction

The purpose of this document is to provide a description of the resource broker architecture as well as providing the client and notification API specifications. The document is thought as a basis for the final architecture documentation, which will be provided as soon as the PM9 will be released.

2. Overview

The Resource Broker (RB) is a middleware that supplies distributed clients with job execution at the more likely Computing Element (CE) in a heterogeneous computing environment. Client applications are provided with a set of API for sending requests and receiving response to/from RB servers. The RB server is responsible for carrying out tasks to satisfy the client requests. These tasks include interacting with the Replica Catalog (RC) to resolve Logical data set names as well as to find a preliminary set of sites where the required data are stored, performing job submission and cancellation by interacting with the Job Submission Service (JSS), listing the more likely resources to execute a job at, and retrieving job outputs on behalf of the clients.

The RB will support GSI (Grid Security Infrastructure) authentication scheme, which is a security infrastructure based on X.509 certificates developed by the Globus group.

The consistency and persistence of all pending jobs information is achieved by means of a data-base (Jobs Registry) local to the RB server itself.

3. Implementation details

The design of the RB server is based on the traditional network connected client/server model. It is composed of two separate servers, RBMaster and RBAgent.

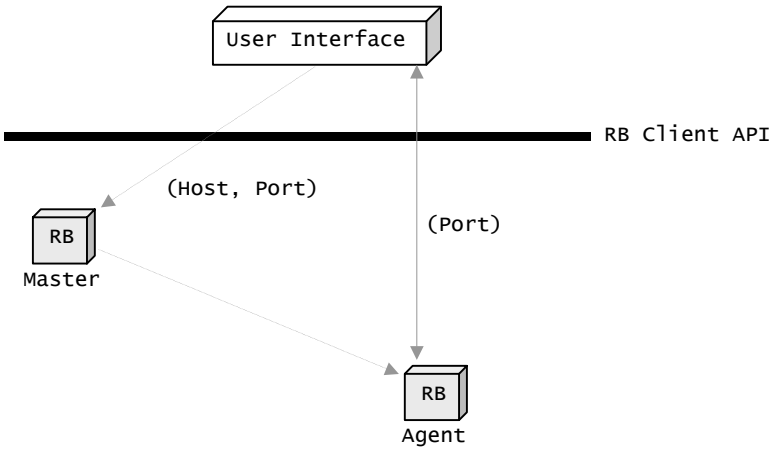

Figure 1 - Resource Broker process model

The RBMaster is the main daemon listening continuously on a well-known port for connection requests from clients. Once a connection from a client is established and authenticated, it execs a RBAgent thread, to service the connection. From that point onward, the client and the RBAgent communicate using a different port and the RBMaster goes back to listening for more connections. Client uses the RB agent to service the desired request.

The User Interface (UI) communicates with the RB agent using a set of API via TCP/IP sockets. The client library sends requests using pre-defined request stubs to the RB agent, and receives and parses replies from the RB agent. The main daemon, the RB master, is responsible for the simple tasks of listening for incoming connections, and spawning a RB agent to handle each connection once the client is authenticated.

The RB will be implemented on Linux RedHat 6.1 for PM9 prototype and will be afterwards also available for Sun Solaris, as well.

3.1. Resource Broker agent design

As described above, the RB is designed based on a multi threading client/server model. Client applications (UI and JSS) are provided with a set of simple API to communicate with the RB servers. The model is distributed in the sense that clients and servers may be running on different hosts.

The RB agents are responsible for receiving and servicing the client request. The RB agents depending on the client request they should accomplish to interact with various entities within the DataGrid network system.

The following picture gives a simplified view of the RB agents design.

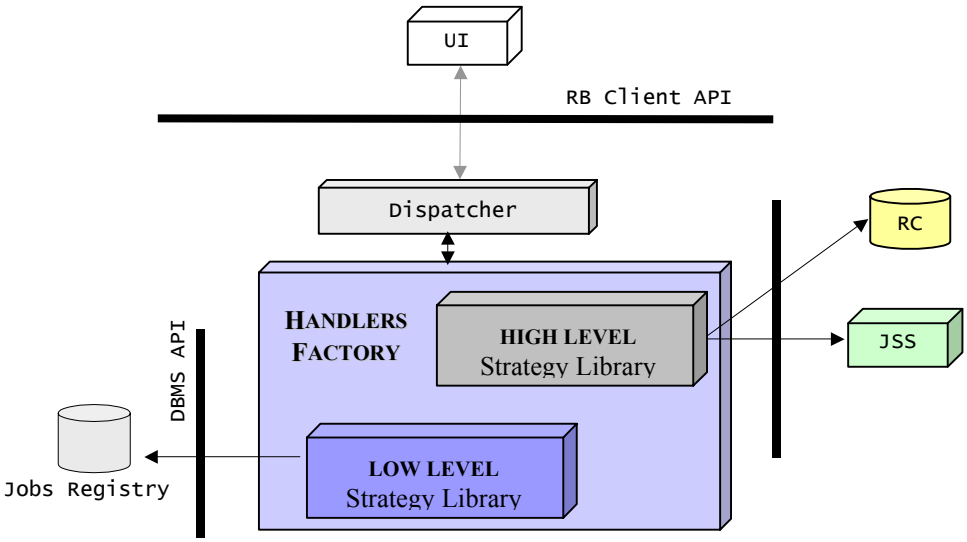

Figure 2 - Resource Broker agent design

At the top is the "dispatcher" module, which listens for incoming client requests and dispatches the requests to the handlers factory. The "dispatcher" is also responsible for returning the results to clients, if so required.

The handler factory of the RB agents in order to accomplish a given request, at run-time configures and runs strategy components depending on the operating phases of that request. The intent is to define a family of algorithms to performs basic functionality, encapsulate each one, and make them interchangeable. Strategy lets the algorithm vary independently from the request that should be accomplished and interchangeable at run-time.

The high-level and low-level handler strategy libraries contains a set of algorithm-objects to be performed in order to accomplish a given client request. The high-level strategy library, contemplate those algorithms which needs interaction with the RC and/or the JSS, whereas the low-level strategy library those which do not need any interaction with the RC and/or JSS.

It is clear that in order to accomplish a given request the RB agent should performs diverse tasks, i.e. resolving logical data set names, finding the resource where the data needed by a given jobs are stored, matching the job requirements, etc. Hard-writing all such algorithms into the classes that require them isn't desirable as different algorithms will be appropriate at different times.

Through the high-level strategies the RB agent accomplishes the following type of requests:

- submit jobs;
- list the more likely resources to execute the job at;
- cancel jobs.

The low-level strategies address low-level algorithms. This module performs the basic I/O operations for moving input/output sandboxes, maintaining/retrieving information about pending jobs.

3.2. Interface to the Job Submission Service

Depending on the client request to be accomplished, i.e. job submission, job cancellation, a request handler should interact with the JSS. It is clear that at this level, as mentioned before, the only interactions regard the job submission and cancellation.

The communication between JSS and RB is achieved by means of API calls. The JSS supply the RB with a set of API calls to full fill job submission and cancellation request, whereas the RB supply the JSS with an API for receiving asynchronous notification from the JSS. It should be pointed out that these notifications are independent to the life cycle of a RB agent. In other words the RB does know neither the instant at which the remote job manager will schedule a submitted job nor when a submitted job completes its execution. Therefore the RB

needs a way to handle such asynchronous notifications issued by the JSS and updating its local jobs registry on reception of such notifications.

To such an aim, it is possible to make the RB master, which is the main broker daemon, listen continuously on a well-known port (different from the one used by the client API) for any call-backs from JSS.

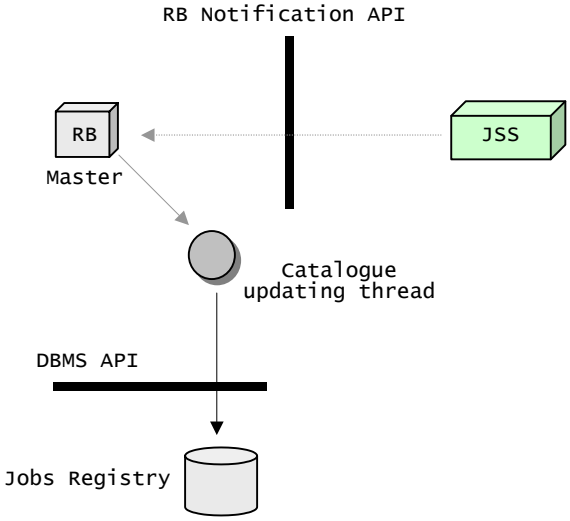

Figure 3 - JSS notifications handling design

As depicted in the previous figure, once the RB master receive a notification from the JSS executes a thread that performs the required update to the jobs registry, maintaining the catalogue coherent to the status of each job.

3.3. Logging and Pruning threads

The main daemon of the RB architecture, as mentioned before, is the RB master, which listen both to the requests from the client and the notifications about job completion, cancellation, and abortion from the JSS. Another task that the RB master daemon should accomplishes is the logging of those events regarding job status. This task is performed by the RB master, which periodically runs a logging thread.

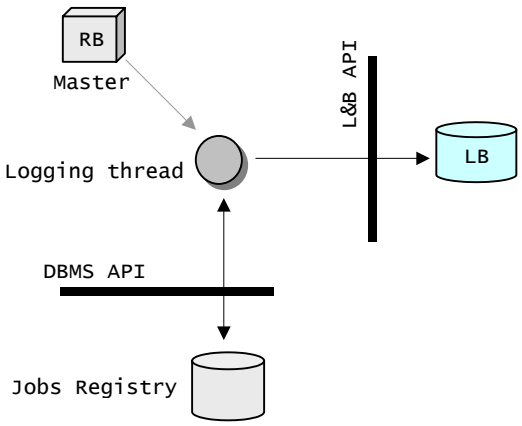

Figure 4 - The logging thread

The idea on the basis of which the logging thread works is straightforward. Once fresh information about pending jobs has been retrieved (i.e. job complete, aborted, etc.), the logging thread by means of the Logging & Bookkeeping API calls sends the appropriate logging event to the LB (i.e. JobTransferEvent, JobAcceptedEvent, JobRefusedEvent, JobAbortEvent).

Another important thread executed by the RB master is the pruning thread. It should be reminded that once a job terminates its execution, the output sandbox is copied back to the storage space of the RB machine, waiting for retrieving following the request of the client. It is clear that such space should be cleared after a given period. A bad user may overload the storage mass capacity by leaving orphan sandboxes at the broker machine. To such an aim, as depicted in the following picture, the RB master periodically executes a pruning thread.

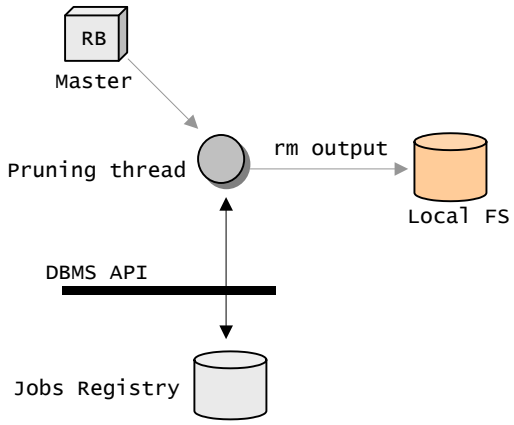

Figure 5 - The pruning thread

The pruning thread by inquiring the job registry acquires information about the time stamp of completed jobs. If the span of time between completion of a job and the current time is greater than a threshold, which should be defined, the output sandbox

of such job will be destroyed and all the information about that job is removed from the registry, as well.

4. Resource Broker Client API.

The Resource Broker Client API supply the client with a single object representing an interface over the services a remote Resource Broker provides for. The full descriptions for each object public method call include a synopsis for the call statement and required Include statements, a description of the functionality, the parameters and return values.

NAME

RB **(class constructor)**

SYNOPSIS

```
#include "rbclient.h"
RB::RB(const string& host, int port);
```

DESCRIPTION

Instantiates an RB server interface for the specified host and port.

- host - The host address of the RB. If the input value is an empty string, it will use the environment variable "RBHost" if it is defined, the hostname of the client machine otherwise.
- port - The port number of the RB. If the input value is negative, it will use the environment variable "RBPort" if it is defined. If not, it defaults to the default port (7771).

RETURN VALUES

None.

ERRORS

None.

NAME

error_message

SYNOPSIS

```
#include "rbclient.h"
const string& RB::error_message();
```

DESCRIPTION

Return the error message from the current client call returned by the RB server. On encountering error on most client calls, in addition to returning an error code, a text string describing the error is returned to the caller. The caller should use the error_message() call to retrieve the error message.

RETURN VALUES

Returns the string containing the error message.

Returns an empty string if there is no error message.

ERRORS

None.

NAME

get_local_logger_contact

SYNOPSIS

```
#include "rbclient.h"

const string& RB::get_LB_contact();
```

DESCRIPTION

Supplies the client with the address of the LB service.

RETURN VALUES

Returns the LB contact string.

ERRORS

In the status member variable of the RB instance:

RB::CONNECTION_ERROR

NAME

job_submit

SYNOPSIS

```
#include "rbclient.h"

void RB::job_submit(const string& jdl);
```

DESCRIPTION

Submit a request to the RB for executing a given job with given requirements at the more likely computing element found.

- jdl – string containing the whole JDL description of the job to be submitted

RETURN VALUES

None.

ERRORS

In the status member variable of the RB instance:

RB::CONNECTION_ERROR

NAME

job_cancel

SYNOPSIS

```
#include "rbclient.h"
void RB::job_cancel(const list<string>& joblist,
     (void* callback)(const string& jobID, bool result));
```

DESCRIPTION

Submit a request for canceling one or more jobs:

- joblist – a list of strings representing the jobID (as generated by the UI) to be cancelled to.
- callback - specifies the procedure-instance address of the client-supplied callback function, which on completion of a job cancellation will receive a string containing the identifier of the cancelled job and a boolean representing the success/failure of such operation.

RETURN VALUES

None.

ERRORS

In the status member variable of the RB instance:

RB::CONNECTION_ERROR, RB::NO_SUCH_JOB

NAME

job_cancel_all

SYNOPSIS

```
#include "rbclient.h"

int RB::job_cancel_all(const string& ucs,
     (void* callback)(const string& jobID));
```

DESCRIPTION

Submit a request to the RB for deleting all the user's jobs.

- ucs - the string containing the user certificate subject.
- callback - specifies the procedure-instance address of the client-supplied callback function that will receive on completion of a successful cancellation the identifier of the cancelled job.

RETURN VALUES

The function returns the number of callbacks the client should expects to, which is equals to the number of jobs to be cancelled.
-1 if an error has occurred while connecting to the resource broker.

ERRORS

In the status member variable of the RB instance:

RB::CONNECTION_ERROR, RB::NO_JOBS_FOR_SUCH_USER

NAME

get_ output_sandbox

SYNOPSIS

```
#include "rbclient.h"
bool RB::get_output_sandbox(const string& jobID, const
string& path);
```

DESCRIPTION

Submit a request to the RB for retrieving all the output files generated by a given job at completion of its execution.

- jobID - the string containing the job identifier as generated by the UI.
- path – a string specifying the local path where the sandbox files should be moved.

RETURN VALUES

Returns whether the output sandbox has been correctly retrieved, or not.

ERRORS

In the status member variable of the RB instance:

RB::CONNECTION_ERROR, RB::NO_SUCH_JOB, RB::NO_SUCH_DONE_JOB

NAME

list_job_matches

SYNOPSIS

```
#include "rbclient.h"
const list<string>&
     RB::list_job_matches(const string& jdl);
```

DESCRIPTION

Submit a request to the RB for displaying the list of identifiers of the queues accessible by the user and satisfying the job requirements included in the job description file.

- jdl - the string containing the job requirements expressed in JDL.

RETURN VALUES

Returns a list of strings containing the resource's identifier matching the job requirements, if any.

ERRORS

RB::CONNECTION_ERROR, in the status field of the RB instance.

5. Resource Broker Notification API.

The Resource Broker Notification API supplies the JSS with a single object representing an interface for sending notifications to a remote Resource Broker. The full descriptions for each object public method call include a synopsis for the call statement and required Include statements, a description of the functionality, the parameters and return values.

NAME

RB **(class constructor)**

SYNOPSIS

```
#include "rbnotify.h"
RB::RB(int port);
```

DESCRIPTION

Instantiates an RB notification interface for the specified port.

- port - The port number of the RB. If the input value in an empty string, it will use the environment variable "RBNotifyPort" if it is defined. If not, it defaults to the default port (9991).

RETURN VALUES

None.

ERRORS

None.

NAME

notify

SYNOPSIS

```
#include "rbnotify.h"
bool RB::notify(RB::notification reason, const string&
jobID,
     const string& remark = "");
```

DESCRIPTION

The JSS announces to the RB the occurrence of an event. The notification events are defined by the enumerative type RB::notification the RB class provides for.

- reason – the event the RB is being notified about. Follows the admissible values and their meaning:
    - JOB_ACCEPTED – the job was accepted;
    - JOB_REFUSED – the job could not be accepted;
    - JOB_ABORTED – the processing of the job was stopped due to system conditions;
    - JOB_DONE – the job has finished its execution;
    - JOB_CANCELLED – the job was successfully removed from the local job manager queue;
    - JOB_OUTPUT_TRANSFERRED – the output files for the job were successfully staged.
- jobID – the string containing the job identifier as generated by the UI.
- remark – optional string describing the reason for notification (e.g. maximum retry count reached, etc.)

RETURN VALUES

Returns whether the RB has correctly received the notification, or not.

ERRORS

None.