# Efficient Grid Service Design to Integrate Parallel Applications

Al. Archip, M. Craus and S. Aruştei

"Gh. Asachi" Technical University of Iasi
Department of Computer Science and Engineering, Romania

**Abstract.** Although grid systems and grid computing have greatly evolved during the past few years, parallel application support remains somewhat limited. A new method for integrating parallel applications as grid services is presented. This method assumes that underlying parallel applications are resources for grid services; also, it implies that service resources may be clients for some predefined helper grid services. The design of the grid service is based on a Factory Service / Instance Service architecture, aiming to offer support for managing multiple resources. The tests were performed on the GRAI Grid (Academic Grid for Complex Applications), using Globus Toolkit 4 – versions 4.0.3 and 4.0.5 – as the base middleware.

## 1 Introduction

The grid computing naturally incorporates parallel applications in order to solve complex problems. This support also includes MPI (Message Passing Interface) based parallel applications. Specialists [1] agree that MPI libraries provide a useful, widely used standard. As presented in [2], Globus Toolkit – including version 4 of the middleware – offers support for parallel application design and implementation through the use of MPICH-G2 package. As depicted in [2] and [3], MPICH-G2 offers full support for MPI v1.1 standard, but also includes some of the more advanced features of MPI v2.0, such as parallel file I/O. While being a platform dependent, MPICH-G2 compensates through speed and through a good integration with Globus middleware.

The highly complex applications, such as data mining or data analysis applications, often require features like the dynamic adding of processing nodes to a local communicator or message exchange between communicators themselves. These features are presented in the MPI-2 standard and are not currently supported by MPICH-G2.

The present paper presents a grid service that wraps around a general MPI application. Through the use of this service, GRAI Grid exposes Data Analysis grid services. Parallelization of the Data Mining application modules has been achieved using LAM 7.1.4.

Archip A., Craus M. and Aruştei S. (2008).
Efficient Grid Service Design to Integrate Parallel Applications.
In *Proceedings of the 2nd International Workshop on Architectures, Concepts and Technologies for Service Oriented Computing*, pages 7-16
DOI: 10.5220/0001898500070016

## 2 Parallel Grid Applications

As presented in the first section, a first possibility of running MPI based applications is to use MPICH-G2. The applications are usually written in C/C++. The code must be compiled with the mpicc (C based applications) or mpic++ (C++ based applications), tools that are integrated in every distribution of MPICH-G2. The general start-up of MPI based applications implemented using MPICH-G2 is given in Fig. 1.

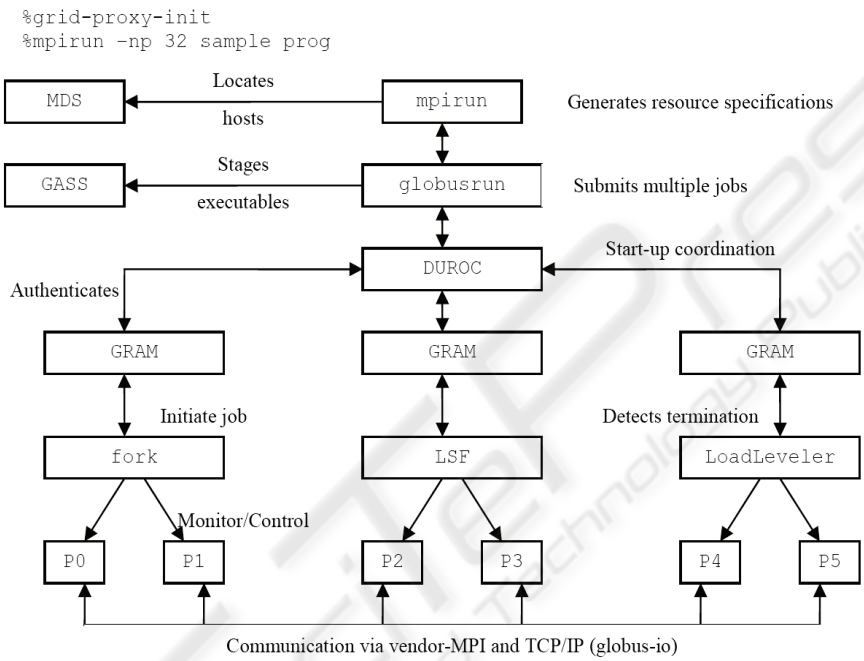


**Fig. 1.** General start-up of MPI-based applications using Globus middleware and MPICH-G2. After a successful grid-proxy-init command, the parallel application may be run either directly, by using the mpirun command, or through the globusrun tool.

An authorized user must perform the following steps in order to run the desired parallel application:

1. The user must check and/or initialize its credentials on the given grid system. Credential checking can be achieved through the grid-proxy-info tool, while credential initialization can be attained through the use of grid-proxy-init tool.
2. After the credentials have been checked/initialized, the user may run the application directly
   (mpirun –np *number_of_processors* path_to_binary), or
3. The user may generate a GRAM RSL script and call the globusrun tool.

If the desired application makes use of standard input or standard output, the above mentioned step 3 is preferred. A sample RSL file (Resource Specification Language)

for GRAM jobs (Globus Resource Allocation Manager) is presented in the following code listing:

```
+( &(resourceManagerContact="frontend.tuiasi.ro")
   (count=4)
   (label="subjob 0")
   (environment=(GLOBUS_DUROC_SUBJOB_INDEX 0)
                (LD_LIBRARY_PATH /opt/globus/lib/))
   (arguments="arg 1" "arg 2" "arg 3")
   (directory="/export/home/alex/testing/mpich_test1")
(executable="/export/home/alex/testing/mpich_test1/a.ou
t")
   (stdout="/tmp/a.out.stdout")
   (stderr="/tmp/a.out.stderr"))
```

This approach has a few major drawbacks. First of all, as stated in section 1, the application does not make use of the full MPI-2 features. Aside from inter/intra-communicator message exchange, C++ applications are not fully supported: MPICH-G2 does not offer support for C++ object serialization/de-serialization. Also, we determined that improper linking of the compiled application may result in GRAM running 4 independent applications instead of one application with 4 processes.

Our approach makes use of WS-GRAM module. For this case, a XML RSL file is presented in the following code listing:

```
<?xml version="1.0" encoding="UTF-8"?>
<job>
<executable>/export/home/alex/testing/mpich_test1/a.out
    </executable>
<directory>/export/home/alex/testing/mpich_test1</direc
tory>
    <argument>”arg 1”</argument>
    <argument>”arg 2”</argument>
    <argument>”arg 3”</argument>
    <stdout>/tmp/a.out.stdout</stdout>
    <stderr>/tmp/a.out.stderr</stderr>
    <count>8</count>
    <jobType>mpi</jobType>
</job>
```

An important note is that, for this second case, the registered user must specify the target cluster and, also, the user must check if the LAM (Local Area Multicomputer) daemon is active in the desired cluster. In order to submit one such job from the command line, the user must issue the following command:

```
globusrun-ws –submit –f rsl.file.name
```

As will be depicted in the following sections, the service we implemented will perform these actions through the use of Java COG Kit (Java Commodity Grid Kit). In order for our service to function properly, the following are required [4]:

- correct installation and configuration of *GridFTP*;
- correct installation and configuration of *ReliableFileTransfer*;
- correct installation and configuration of *ManagedJobFactoryService*;
- adequate firewall configuration that would allow for TCP ports 2811 and 8443 to accept connections from authorized hosts.

# 3 Grid Service Architecture

The architecture of our grid service is compliant with the *Factory Service* / *Instance Service* model submitted in [5] (see Fig. 2). Such a model allows for a good management of resources and offers good support for multiple users accessing the same service.

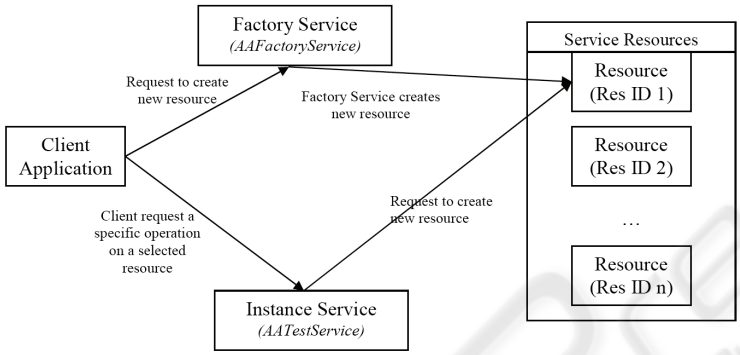


**Fig. 2.** Factory Service / Instance Service model. A client may connect either to Factory Service and create a new WS-Resource for a new analysis, or to a previously initialized WS-Resource through Instance Service.

## 3.1 Grid Service Implementation

Using the model in Fig. 2, our service is divided into two major components. The first one is the Factory component, *AAFactoryService*. The implementation for this component is similar to the one depicted in [5], without any relevant changes. *AAFactoryService* has only a basic functionality implemented: its single purpose is to create new resources to be used by the second component, *AATestService*. Once a WS-Resource is properly created, *AAFactoryService* returns an *EndpointReferenceType* that uniquely identifies the newly created WS-Resource.

The second major component, *AATestService*, is responsible for the data analysis itself. It has only four methods accessible to the client:

- *methodWriteSettings* – this method should be used to write configuration data needed for the analysis (being a data mining task, configuration data must include parameters like minimum support, minimum confidence, target database and means of connecting to the given database);
- *methodWriteLogSettings* – this method is specific to our tests (as will be detailed in section 4 of this paper).
- *methodClientStartJob* – this is the main method of the *Instance service*. The method builds the XML RSL file and then calls the current WS-Resource for job submitting;
- *methodReadJobStatus* – once a job is submitted to run, the client may check on the status of the current job by calling this method.

An interesting aspect of our service is related to the WS-Resource it relies on. After a correct initialization – through calling the Factory Service and the through setting the appropriate analysis parameters – the resource would act, on behalf of the authorized user, as a client for the *ManagedJobFactoryService* provided by the Globus middleware. A succinct listing of our resource implementation code is given:

```
package aacore.factory.impl;
//for job submittion
import gridUtils.MyGridProxy;
import gridUtils.WSJobWrapper;
import org.ietf.jgss.GSSCredential;/*Common imports
omitted*/
public class AATestResource implements Resource,
ResourceIdentifier, ResourceProperties {
  private ResourcePropertySet propSet;
  private Object key;
  private String jobSettingsRP;
  private String jobLogPrefixRP;
  private String clientJobStatusRP;
  public Object initialize() throws Exception {}
  public String getJobSettingsRP() {}
  public void setJobSettingsRP(String value) {}
  public String getJobLogPrefixRP() {}
  public void setJobLogPrefixRP(String value) {}
  public String getClientJobStatusRP() {}
  public void setClientJobStatusRP(String value) {}
  public ResourcePropertySet getResourcePropertySet(){}
  public Object getID() {}
  public int runJob(String  jobRSL, String proxyPath,
         String keyFile, String certFile) {}
}
```

The main method of the above listed class is *runJob*. This method uses two personalized classes – *MyGridProxy* and *WSJobWrapper* – in order to connect to *ManagedJobFactoryService*. The method updates the private variable *clientJobStatusRP* with the status results received for the current job. Each resource identifies a single job and each resource is identified by a corresponding *EndpointReferenceType* – previously returned by our *AAFactoryService*. Through specific security settings (as depicted in [6]), self authorization is achieved and the WS-Resource submits the current job with authorized user's credentials. The client – as section 3.2 will describe – is responsible for proxy initialization/re-initialization.

### 3.2 Client Description

The message exchange between our Grid client and the Grid service is presented in fig. 3.

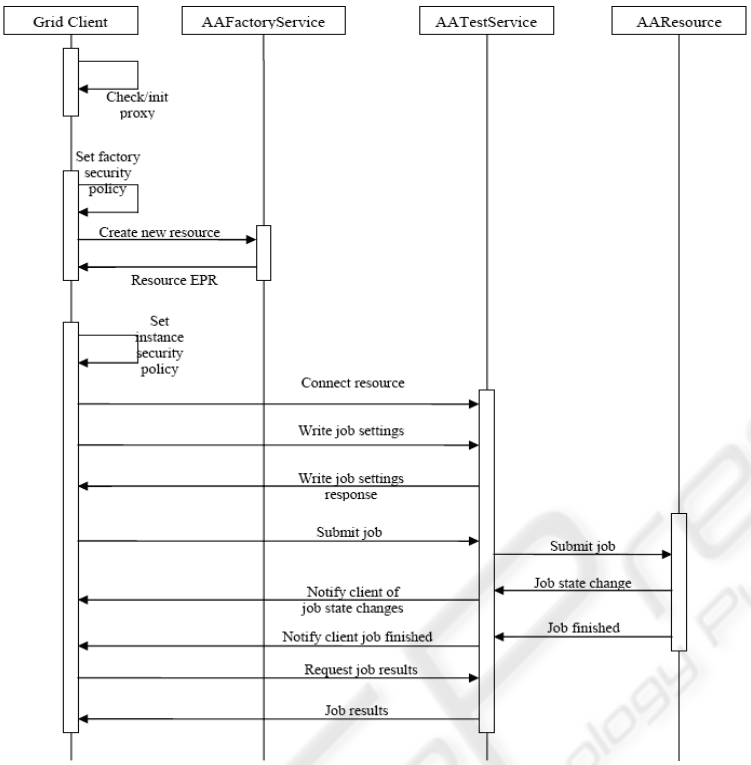


**Fig. 3.** Message exchange between the Grid client, Factory Service and Instance Service. The Instance Service receives notifications from the *ManajedJobFactoryService* and then notifies the client on job state changes.

A client for our service must perform the following tasks:

- Check the user proxy – if no valid proxy is found or if the previous proxy has expired, the client must initialize/re-initialize the proxy.
- Check for existing resources – this is performed through a set of specialized classes (described in section 3.3); if previous resources are found, the client may check the job status of the respective resources.
- Initialize a new resource for a new job submitting or connect to an existent resource for job monitoring – the client is responsible for submitting the correct credentials to the service.
- When the current analysis is done and no other tasks have been submitted, the client must destroy his/her proxy.

Both the client and the service require a set of specialized classes to work. These classes are described in the following section.

### 3.3 Service/Client Specific Components

Since both the client and the service rely on user authorization, the following code listing describes the *MyGridProxy* class – a class that is used in user proxy management:

```
package gridUtils;
/*Common imports omitted*/
import fileHelpers.FilePermHelper;
public class MyGridProxy {
     private X509Certificate certificate;
     private PrivateKey userKey = null;
     private GlobusCredential proxy = null;
     private ProxyCertInfo proxyCertInfo = null;
     private int bits = 512;
     private int lifetime = 3600 * 12;
     private int proxyType;
     private GlobusGSSCredentialImpl credimpl;
     private String proxyFile;
     private String keyFile;
     private String certFile;
     private String issuer;
     private String user, password;
     public MyGridProxy() {}
     public MyGridProxy(String proxyFile,
              String keyFile, String certFile) {}
     public void environmentSetup() {}
     public void createProxy() throws Exception {}
     public boolean checkProxy() {}
     public boolean destroyProxy() {}
     public GSSCredential buildProxy() {}
     public GSSCredential buildProxy(byte[]proxyData){}
}
```

The client must use the *checkProxy* and/or *createProxy* methods for proxy initializations. The service must use the specific *buildProxy(byte[] proxyData)* method in order to check for a valid user proxy and submit or monitor the job.

Aside from the steps described in [7], we have implemented a second helper class that would set the appropriate file permissions for the proxy file. On a Linux platform, these permissions must be *read/write* for the user and no permissions for the user's group and for the others. Since FileWriters in Java create files with supplementary permissions (read permissions for both user's group and for other users), a helper class (*FilePermHelper*) has been developed in order to set the appropriate file rights.

In order for the Resource to correctly submit a job, we have implemented a WS-Gram Job wrapper. This class is used only by the service and has the purpose of contacting the corresponding *ManagedJobFactoryService*. A brief code listing follows:

```
package gridUtils;/*Common imports omitted*/
public class WSJobWrapper implements GramJobListener {
     private String rslFileName;
     private GramJob crtJob;
```

```
        private static final long
             STATE_CHANGE_BASE_TIMEOUT_MILLIS = 10000;
        private boolean jobCompleted;
        private GSSCredential proxy;
        private int exitCode;
        public void setRSL(String rslFileName) {}
        public String getRSL() {}
        public void setProxy(GSSCredential proxy) {}
        public GSSCredential getProxyPath() {}
        public boolean jobDone() {}
        public WSJobWrapper() {}
        public WSJobWrapper(String rslFileName,
                 GSSCredential proxy) {}
        public void stateChanged(GramJob job) {}
        public void submitRslFile() {}
        public int processCrtJob(GSSCredential proxy,
                EndpointReferenceType factoryEPR){}
        private synchronized void waitJobCompletion() {}
    }
```

An important note is that, if the resource does not receive a state change notification for a predefined period of time, it will assume job completion and will notify the corresponding service of job completion. Although this is not the preferred method for job monitoring, it does not lock computational resources indefinitely.

In order to achieve multiple resource management abilities the client uses two more classes named *EPRFileHelper* and *EPRFileLister*, respectively. The first class is used for saving the endpoint reference returned by *AAFactoryService*. The client may then connect either to a newly created resource in order to submit a new job or to an existing resource in order to monitor a specific job. The second class is used to monitor a client specified folder for endpoint references.

## 4 Case Study

The above described grid service is currently being tested on GRAI Grid as a wrapper service for various data mining algorithms. One such study involves the implementation of Hash Partitioned Apriori (HPA). We have implemented the algorithm using C++ STL. Parallelization has been implemented using:

1. MPICH-G2 – the standard version embedded in the Rocks version 4.2.1 Linux distribution.
2. MPI 2.0 compliant libraries implemented in LAM 7.1.1 (using the same Linux distribution as support operating system).

In order to properly monitor the submitted jobs, the test application used a custom *log* module. Each processing node writes detailed information about the current step of the analysis in a specific log file. This method is preferred due to an increased control over processes and thorough debug in case of failure. Test job has been submitted using first time four and then eight processes.

For case 1) we used both GRAM and WS-GRAM approaches. The grid service submitted the job without any errors and received correct notifications only when using GRAM scripts. However, the GRAM approach failed to initialize properly,

submitting four/eight independent applications instead of a single application with four/eight concurrent threads. Since Globus Toolkit specifications maintain GRAM job submitting only for backward compatibility issues, this first case is not preferred for further use.

For case 2), we used only the WS-GRAM approach and we have manually started the LAM daemon prior to running the service. The job submitting worked as expected. The grid service performed accurate monitoring only after appropriate security settings have been employed: the *methodClientStartJob* method of the grid service must be configured to run with user credentials instead of the default container credentials.

This run proves one of the main advantages of our service. We do not limit the development of parallel applications only to MPICH-G2.

The MPI application that we have used for tests required some specific configuration files. A usual scenario for running MPI-based Grid applications that require file upload and/or file download. Since MPI-based applications have poor or no support for interacting with Grid services, the user had to manually upload various configuration and input files. In our case the Instance service connects directly to the ReliableFileTransfer service and transfers any needed files between the client and the MPI application. This shows a second advantage of our service: the service, acting as a job wrapper, may interact with various other Grid services before or after submitting the MPI job.

## 5 Conclusions and Further Research

The present paper describes a new way of combining Java Grid applications with MPI based parallel applications into a Factory/Instance grid service. The need for such a grid service is derived from the fact that current grid middleware support for MPI based parallel applications do not fully use MPI 2.0 standard. Also, MPICH-G2 or other MPI standard implementations do not allow for a direct connection between parallel applications and middleware grid services (such as OGSA-DAI or ReliableFileTransfer service).

The tests we performed using a specific console client have demonstrated the usefulness of the presented model in managing multiple resources, as well as a good interaction with several grid services delivered with Globus Toolkit 4 middleware (services such as *ReliableFileTransfer* or *ManagedJobFactoryService*). The tests have shown that correct use of WS-GRAM RSL scripts combined with appropriate security settings for the corresponding grid service prove useful in embedding MPI based parallel applications within grid services.

Aside from the two major advantages pointed out in section 4, we would also like to point out another plus for our model. The service's Resource specifies the MPI modules it uses in a static manner. While this is not the preferred method for a general wrapper application, it has the advantage of not submitting foreign, unchecked code on our Grid system.

We are currently investigating means of pipe-lining specific application logs with Java applications in order to increase the level of detail for grid job status. A positive result would achieve better job management through the grid services.

## Acknowledgements

The Excellence Research Program, through grant 74 CEEX-II03 – "Academic Grid for Complex Applications", has supported the research for this paper.

## References

1. Shantenu Jha, Grid Experts Address Barriers to Distributed Applications, http://www.gridtoday.com/grid/1735208.html, August 20, 2007
2. MPICH-G2, http://www.globus.org/grid_software/computation/mpich-g2.php
3. Nicholas T. Karonis, Brian Toonen, Ian Foster, MPICH-G2: A Grid-Enabled Implementation of the Message Passing Interface, November 2002
4. J. Bart, M. Brown, K. Fukui, N. Trivedi, Introduction to Grid Computing, IBM RedBooks, published: 27 December 2005, http://www.redbooks.ibm.com/abstracts/sg246895.html?Open
5. Sotomayor B., The Globus Toolkit 4 Programmer's Tutorial, http://gdp.globus.org/gt4-tutorial/multiplehtml/ch05s01.html, 2005
6. Globus Toolkit Alliance, Security Descriptors, http://www.globus.org/toolkit/docs/4.0/security/authzframe/security_descriptor.html
7. Globus Toolkit Alliance, Submitting a job in Java using WS GRAM, http://www.globus.org/toolkit/docs/development/4.1.0/execution/wsgram/wsgram-scenarios-java.html