



Requirements and Solutions Survey: Initiator-Locator-Servent Model

Project Title: PGPGrid

Document Title: Requirements and Solutions Survey: Initiator-Locator-Servent Model

Document Identifier: EPCC-IS-PGPGRID-REQ-ILS-1.0

Distribution Classification: Public

Authorship: Paul J Graham

Approval List: PGPGrid Team

Distribution List: Public

Document History:

| Personnel | Date | Summary | Version |
|---------------|------------|---------------|---------|
| Paul J Graham | 27/04/2004 | First release | 1.0 |

| | | |
|-----|------------------------------|----|
| 1 | Introduction | 3 |
| 2 | Requirements..... | 3 |
| 2.1 | Register..... | 3 |
| 2.2 | Add Servent | 4 |
| 2.3 | Request Resource | 4 |
| 2.4 | Request Job Run | 4 |
| 2.5 | Update Resource Status | 5 |
| 2.6 | Submit Job | 5 |
| 2.7 | Job Completed | 5 |
| 2.8 | General | 6 |
| 2.9 | Aggregation | 6 |
| 3 | Related technologies..... | 8 |
| 3.1 | Condor | 8 |
| 3.2 | Web Services..... | 10 |
| 3.3 | EPCC ePortal..... | 12 |
| 3.4 | Grid Services | 12 |
| 3.5 | GRAM | 13 |
| 3.6 | JOSH | 15 |
| 3.7 | UNICORE | 17 |
| 3.8 | JCSP | 19 |
| 3.9 | CSF | 19 |
| 4 | Recommendations | 19 |
| 5 | References | 20 |

1 Introduction

This document identifies some of the requirements for the Initiator-Locator-Servent (ILS) model discussed in the Use Case document “Use Cases: Initiator-Locator-Servent model” [1] as part of the PGPGrid project workpackages 4 and 5. The document also includes a short survey to attempt to identify some of the existing technologies that may satisfy many, if not all of the requirements.

2 Requirements

In this section we shall list the key requirements derived from each Use Case in [1]. Each requirement will be defined as a *must* or a *should*, where *musts* are the minimum required functionality for a working ILS model, and *shoulds* will expand the functionality but are not absolutely necessary. That is not to say that the *shoulds* will be ignored but this categorisation will hopefully help to prioritise some of the functionality. The requirements will then be aggregated into appropriate groupings.

2.1 Register

The Servent needs to contact the Locator:

- There *must* be a mechanism for the Servent to find a Locator.
- There *should* be a mechanism for the Servent to find multiple Locators – it is conceivable that there may be situations where the Servent wishes to register itself with several different Locators.

The Servent and Locator need to communicate:

- There *must* be a mechanism to allow communication between the Servent and the Locator – information needs to be exchanged.
- There *should* be a mechanism to allow *secure* communication between the Servent and the Locator – for certain uses of the ILS model there may be a requirement for secure communication.
- The mechanism for communication *should* operate across different organisations – the intention of the ILS model is to allow resources at various institutions to be utilised seamlessly.
- The mechanism for communication *should* operate across different platforms - it is hoped that the ILS model will allow the exploitation of resources with various OS. **NB** one could argue, in the context of PGPGrid, that this and the previous requirement with regard to communication should be *musts* rather than *shoulds*. However, they are listed here as *shoulds* to imply they need not be in the first set of requirements to be tackled by any solution. That is, we should concentrate on getting the communication infrastructure working within a simple test case first, yet bear in mind the future usage scenarios for communications between disparate institutes.
- All components (Servent, Initiator, Locator) *must* be uniquely identifiable – it is likely that there will be many instances of each component and there cannot be confusion as to which instance is being referred to.

The Servent needs to register with the Locator:

- The Servent *must* supply a set of static parameters describing the underlying resource it represents – to be used in resource matching (Section 2.3).
- The Servent *must* have access to static information regarding the underlying resource it represents – this includes such things as processor speed, memory etc (see [1] for the agreed list).

2.2 Add Servent

The Servent registers itself with the Locator:

- The Locator *must* have a visible method for a Servent to register itself.
- The Locator *must* maintain a list of registered Servents – to prevent a Servent registering itself multiple times and for reference purposes.
- The Locator *must* be capable of registering many Servents.
- The Locator *must* maintain the parameter information for each Servent – this is required for resource matching to jobs (see Section 2.3).

2.3 Request Resource

The Initiator needs to specify parameters for a job:

- A set of parameters *must* be defined for each job to be run within the model – these parameters will include such things as required CPU time, memory etc.
- The Initiator *must* have a mechanism for the job run parameters to be specified.
- The Initiator *should* allow ranges to be set for the job parameters where appropriate.

The Initiator needs to pass job parameters and request for resource to the Locator:

- The Initiator *must* know of the Locator – perhaps a lookup table.
- The Initiator *should* know of multiple Locators – the Initiator may want access to more resource than is available via one Locator.
- The Initiator and Locator *must* have a mechanism for communication – same as the requirements for Servent-Locator communication in Section 2.1.
- The Locator *must* have a visible method to allow submission of a resource request.
- The Initiator *must* compose a resource request consisting of amount of resource required and job parameters – for example ten processors for ten jobs which each must have a certain processor speed, memory etc.

The Locator needs to match the job parameters against the available resources:

- The Locator *must* have an algorithm for matching jobs to resources.
- The Locator *should* have an algorithm for ranking appropriate resources by ‘suitability’ for a job.
- The Locator *must* return a list of available resources that fit the job criteria.
- The Locator *should* return a ranked list of available resources – this should be based on how best the resources fit the criteria, for example faster resources first.
- The returned list of available resources *must* contain enough information for the Initiator to contact the Servent directly.

There may be no suitable resources:

- The Locator *must* inform the Initiator if there are not enough available resources to match the request.
- The Locator *should* inform the Initiator of why there are not enough available resources – if it is because they are busy it may be worth submitting the request at a later date, but if none of the resources are capable of running the job then a different course of action has to be taken.
- The Initiator *must* handle the lack of suitable resources situation – some of the possibilities are discussed further in [1].

2.4 Request Job Run

Initiator negotiates with a Servent for use of a resource:

- There *must* be a mechanism for the Initiator and Servent to communicate – similar to the requirements for Servent-Locator communication in Section 2.1.
- The Servent *must* provide a method for requesting resource.
- The Initiator *must* submit a resource request to the Servent.
- The Initiator *should* include job parameters in the resource request – it is conceivable that at some point the Servent (and its underlying resource) may require this information in order to decide to accept the job or not.
- The Servent *must* decide whether or not to grant the resource request.
- The Servent *should* decide whether or not to grant the resource request based on the job parameters.
- The Servent *should* have some mechanism for querying the underlying resource to see if it will accept the job.
- The Servent *must* return the decision to the Initiator.

Initiator responds to a negative decision (again, details discussed in [1]):

- Initiator *must* act on a negative decision.
- Servent *should* supply Initiator with reason for the negative response.
- Initiator *should* submit resource request to next ranked Servent.
- Initiator *should* attempt to contact a different Locator with the resource request.

Initiator responds to a positive decision: see Section 2.6.

2.5 Update Resource Status

The Locator needs to know the status of the Servent's underlying resource:

- There *must* be a method to allow the resource status to be updated – this could be a pull (i.e. the Locator queries the Servent) or more likely a push (the Servent sends status messages to the Locator) model.
- The Locator *must* maintain dynamic information for the resource – such as number of processors available.

2.6 Submit Job

The job needs to be able to run on the remote resource:

- The job *must* be a Java executable - a JAR file.

The job needs to be sent to the Servent:

- The Servent *must* have a public method for receiving jobs.
- The Servent *must* have access to the executable JAR – the JAR itself may be sent to the Servent, or a URL indicating its location to the Servent so it can be retrieved
- The Servent *must* return a unique identifier for the job – as discussed in [1], there may be several jobs from the same Initiator.

The job needs to run on the resource:

- The Servent *must* be able to submit the job to run on its underlying resource.
- The Servent *should* update the Locator with its status.
- The Servent *should* be able to monitor the jobs progress.
- The Servent *should* have a mechanism for notification of when the job has completed.

2.7 Job Completed

The Initiator must be informed of the job completion:

- There *should* be a public method allowing the Initiator to be informed of a job's completion.
- There *should* be some statistics produced and stored for the job execution.
- There *should* be some metrics analysed (such as data I/O) to measure successful completion of the job – the job may have finished but it may be necessary to confirm it has completed successfully by, for example, using checksums.

2.8 General

- The Servent, Initiator and Locator *should* be able to run on any platform.
- The Servent, Initiator and Locator instances *must* each be uniquely identifiable.
- There *must* be some fault tolerance at all stages of the system – this is a difficult requirement to specify in detail. Clearly there are many points of possible failure, such as in communications or component persistence. However, it should be kept in mind as a requirement that applies to all aspects of any software developed or utilised in the model.

2.9 Aggregation

In this section we attempt to group the requirements into similar areas and number them for future reference. Note that for several of the 'external interactions' one component having a requirement sometimes implies another component has a corresponding requirement but is not explicitly defined. For example, the Initiator has a requirement for including job parameters in a resource request, which implies that there is a corresponding request mechanism in the Servent.

2.9.1 Look up

| Number | Requirement |
|--------|---|
| 1.1 | There <i>must</i> be a mechanism for the Servent to find a Locator |
| 1.2 | There <i>should</i> be a mechanism for the Servent to find multiple Locators |
| 1.3 | All components (Servent, Initiator, Locator) <i>must</i> be uniquely identifiable |
| 1.4 | The Initiator <i>must</i> know of the Locator |
| 1.5 | The Initiator <i>should</i> know of multiple Locators |
| 1.6 | The returned list of available resources <i>must</i> contain enough information for the Initiator to contact the Servent directly |

2.9.2 Communications

| Number | Requirement |
|--------|---|
| 2.1 | There <i>must</i> be a mechanism to allow communication between the Servent and the Locator |
| 2.2 | There <i>should</i> be a mechanism to allow <i>secure</i> communication between the Servent and the Locator |
| 2.3 | The mechanism for communication <i>should</i> operate across different organisations |
| 2.4 | The mechanism for communication <i>should</i> operate across different platforms |
| 2.5 | The Initiator and Locator <i>must</i> have a mechanism for communication |
| 2.6 | There <i>must</i> be a mechanism for the Initiator and Servent to communicate |

2.9.3 Locator external interaction

| Number | Requirement |
|--------|-------------|
|--------|-------------|

| Number | Requirement |
|--------|---|
| 3.1 | The Locator <i>must</i> have a visible method for a Servent to use to register itself |
| 3.2 | The Locator <i>must</i> have a visible method to allow submission of a resource request |
| 3.3 | The Locator <i>must</i> return a list of available resources which fit the job criteria |
| 3.4 | The Locator <i>should</i> return a ranked list of available resources |
| 3.5 | The Locator <i>must</i> inform the Initiator if there are not enough available resources to match the request |
| 3.6 | The Locator <i>should</i> inform the Initiator of why there are not enough available resources |

2.9.4 Locator internal operation

| Number | Requirement |
|--------|--|
| 4.1 | The Locator <i>must</i> maintain a list of registered Servents |
| 4.2 | The Locator <i>must</i> be capable of registering many Servents |
| 4.3 | The Locator <i>must</i> maintain the parameter information for each Servent |
| 4.4 | The Locator <i>must</i> have an algorithm for matching jobs to resources |
| 4.5 | The Locator <i>must</i> maintain dynamic information for the resource |
| 4.6 | The Locator <i>should</i> have an algorithm for ranking appropriate resources by 'suitability' for a job |

2.9.5 Servent external interaction

| Number | Requirement |
|--------|--|
| 5.1 | The Servent <i>must</i> provide a method for requesting resource |
| 5.2 | The Servent <i>must</i> return the resource request decision to the Initiator |
| 5.3 | The Servent <i>should</i> supply Initiator with reason for the negative response to the resource request |
| 5.4 | The Servent <i>must</i> have a public method for receiving jobs |
| 5.5 | The Servent <i>must</i> return a unique identifier for the job |
| 5.6 | The Servent <i>should</i> update the Locator with its status |
| 5.7 | The Servent <i>must</i> have access to the executable JAR |

2.9.6 Servent internal operation

| Number | Requirement |
|--------|---|
| 6.1 | The Servent <i>must</i> supply a set of static parameters describing the underlying resource it represents |
| 6.2 | The Servent <i>must</i> have access to static information regarding the underlying resource it represents |
| 6.3 | The Servent <i>must</i> decide whether or not to grant the resource request |
| 6.4 | The Servent <i>should</i> decide whether or not to grant the resource request based on the job parameters |
| 6.5 | The Servent <i>should</i> have some mechanism for querying the underlying resource to see if it will accept the job |
| 6.6 | The Servent <i>must</i> be able to submit the job to run on its underlying resource |
| 6.7 | The Servent <i>should</i> be able to monitor the jobs progress |
| 6.8 | The Servent <i>should</i> have a mechanism for notification of when the job has completed |

2.9.7 Initiator external interaction

| Number | Requirement |
|--------|---|
| 7.1 | The Initiator <i>must</i> have a mechanism for the job run parameters to be specified |

| Number | Requirement |
|--------|---|
| 7.2 | The Initiator <i>should</i> allow ranges to be set for the job parameters where appropriate |
| 7.3 | The Initiator <i>must</i> submit a resource request to the Servent |
| 7.4 | The Initiator <i>should</i> include job parameters in the resource request |
| 7.5 | The Initiator <i>should</i> have a public method allowing it to be informed of a job's completion |

2.9.8 Initiator internal operation

| Number | Requirement |
|--------|---|
| 8.1 | The Initiator <i>must</i> compose a resource request consisting of amount of resource required and job parameters |
| 8.2 | Initiator <i>must</i> act on a negative decision for a resource request |
| 8.3 | The Initiator <i>must</i> handle the lack of suitable resources situation |
| 8.4 | Initiator <i>should</i> submit resource request to next ranked Servent |
| 8.5 | Initiator <i>should</i> attempt to contact a different Locator with the resource request |

2.9.9 Other

| Number | Requirement |
|--------|--|
| 9.1 | A set of parameters <i>must</i> be defined for each job to be run within the model |
| 9.2 | There <i>must</i> be a method to allow the resource status to be updated |
| 9.3 | The job <i>must</i> be a Java executable |
| 9.4 | There <i>should</i> be some statistics produced and stored for the job execution |
| 9.5 | There <i>should</i> be some metrics analysed (such as data I/O) to ensure successful completion of the job |
| 9.6 | The Servent, Initiator and Locator <i>should</i> be able to run on any platform |
| 9.7 | The Servent, Initiator and Locator instances <i>must</i> each be uniquely identifiable |
| 9.8 | There <i>must</i> be some fault tolerance at all stages of the system |

3 Related technologies

In this section we shall examine some existing technologies and investigate their appropriateness with respect to the requirements in section 2.9.

3.1 Condor

From the Condor website [2]:

Condor is a specialized workload management system for compute-intensive jobs. Like other full-featured batch systems, Condor provides a job queuing mechanism, scheduling policy, priority scheme, resource monitoring, and resource management. Users submit their serial or parallel jobs to Condor, Condor places them into a queue, chooses when and where to run the jobs based upon a policy, carefully monitors their progress, and ultimately informs the user upon completion.

In general, Condor is aimed at utilising spare cycles from an organisation's computing infrastructure, so-called *opportunistic* batch processing. It also provides support for job matching with resources and supports more advanced features such as checkpointing and job migration. A key feature which would potentially be useful to Glasgow's 3D visualisation processing is the *DAGMan* utility, which manages the submission of a large number of jobs with possibly complex dependencies on each other. Condor also provides support for accessing resources that are using Globus via Condor-G, and across disparate organisations via their *flocking* utility.

It can be envisaged that Condor would take the place of the Servent and Locator in our model, and some interface between Condor and the Initiator would need to be developed. Table 1 below shows how Condor matches up to the requirements.

3.1.1 Condor: summary

Table 1: Condor and the requirements

| Requirement | Condor |
|-----------------------------------|---|
| 1. Look up | All but one of these requirements are covered by Condor. What Condor does not support is the Initiator-Servent look up, as within the Condor model there is no direct interaction between the job submitter and the resource: it is all done via the Locator. |
| 2. Communications | <p>The user submitting a job handles Initiator-Locator communications. Locator-Servent is handled within the Condor system. It is not clear if any of this is secure, although the Condor-G system exploits Globus security.</p> <p>For communications across separate organisations there is Condor-G for Globus-enabled resources, or <i>flocking</i>, for multiple independent Condor pools. For cross-platform, Condor supports a broad range of Windows and Unix environments (see [5] for an exhaustive list).</p> <p>There is no direct communication between Initiator and Locator.</p> |
| 3. Locator external interaction | <p>Installing the appropriate Condor binaries, setting up configuration files and starting the daemons is the equivalent of a Servent-Locator registration. Submission is done via job submission files and command-line arguments.</p> <p>As for returning available resources, essentially the Condor model is set up for the job to be submitted to the Condor server and it handles the job matching and submission. This raises an issue with the JPie model as the Initiator needs to know where the job will run to set up the correct data transfer pipes. It is possible to check where the job is running once it has been submitted, but this is too late for JPie.</p> <p>Condor holds submitted jobs in a queue until a resource becomes available. The Initiator can check the job status via the command line.</p> |
| 4. Locator internal operation | This is where Condor performs well as it has a comprehensive job to resource matching system. Daemons running on the resources maintain the dynamic information. |
| 5. Servent external interaction | Here the Servent-Locator interactions are handled by the Condor daemons, however there are no Servent-Initiator interactions. |
| 6. Servent internal operation | The Condor server daemon appears to handle all of these requirements. |
| 7. Initiator external interaction | The Initiator would have to be written to use the command-line arguments to interact with Condor. Condor sends an email on job completion; otherwise the Initiator would have to actively query Condor for the job status. |
| 8. Initiator internal operation | The resource operations are handled by the Condor daemons. The request composition would use the well defined Condor job submit description files. |

| Requirement | Condor |
|-------------|---|
| 9. Other | Condor provides job parameter definitions. There is support for Java executables. Condor provides Servent and Locator portability and identity. Condor appears to be reasonably fault tolerant. |

Condor is designed for a generic compute environment and, as such, satisfies many of our requirements with little effort on our behalf. However, it does not provide for Initiator-Servent communication and the necessary ‘where will this job be running’ knowledge for the JPie model. It may be possible to get around this (perhaps with some sort of ‘placeholder’ job). Any Initiator-Locator communication is performed via the command-line. It also requires a significant installation procedure at each resource, and prefers to be the only batch system installed on a resource, which will generate some issues.

For the “job-to-resource” matching, the Locator-Servent interactions and for job submission, Condor would appear to be a good solution. Also the Servent/underlying resource interactions are dealt with in a reasonable manner. Condor is a mature, well-established technology with all that entails. However, it is not clear that it fits well into the JPie-determined model that we require, due to the lack of Initiator-Servent direct interaction.

3.2 Web Services

Web Services (WS) have arisen from the need for application-to-application communication over the World Wide Web. They are the programmatic interfaces that are made available across the WWW. In a typical web services scenario, an application sends a request to a service at a given URL using the SOAP (Section 3.2.1) protocol over HTTP. The service receives the request, processes it, and returns a response. Services are usually described using the Web Services Description Language (see Section 3.2.2). XML is the underpinning technology for utilising Web Services.

3.2.1 SOAP

The Simple Object Access Protocol (SOAP [6]) is a lightweight protocol for exchange of information in a decentralised, distributed environment. From the W3C [7]:

It is an XML based protocol that consists of three parts: an envelope that defines a framework for describing what is in a message and how to process it, a set of encoding rules for expressing instances of application-defined datatypes, and a convention for representing remote procedure calls and responses.

So it is a simple, one-way message exchange paradigm. It is used in Web Services as the method of communication. As well as employing HTTP it can also operate over other transport protocols such as FTP or SMTP.

3.2.2 WSDL

The Web Services Description Language [8] is an XML schema that specifies the format of XML documents for describing a Web Service. A WSDL document defines a number of elements that contribute to the interface description:

- Input and output messages – analogous to argument and return types.
- Operations defined from input and output messages – analogous to methods or functions.
- Operations are then aggregated into *portTypes* which are application-specific collections of related operations – for example operations for digital signal processing or managing a registry.

Each portType is then combined with a network protocol (e.g. HTTP) and message format (e.g. SOAP) to form a *binding*, which describes how to communicate with the service. Each binding is associated with a specific network address to form a *port*, which describes where to find the service. Finally, a collection of ports is aggregated into a *service*.

3.2.3 Java Web Services

For many programmers Java is the natural choice for developing WS as it is highly portable and, by their very nature, WS are distributed on different computing platforms. Java also has extensive API support for handling XML. In addition, the Java 2 Platform Enterprise Edition (J2EE) is specifically designed to assist in developing and deploying web services. Also of interest to us is the Java API for XML registries (JAXR) that eases access to XML registries and thus, for example, assists in the discovery of web services.

3.2.4 Other Web Services Implementations

Of course, the web services implementation does not necessarily have to be Java-based. One of the staple ideas of WS is platform and language independence. For example, Microsoft's C# programming language has excellent support for developing web services and there are several C# compilers for a range of platforms. Also there is considerable expertise at EPCC in developing C# web services (mainly through the MS.NETGrid project [4]).

3.2.5 WS: Summary

Table 2: Web Services and the requirements

| Requirement | Web Services |
|-----------------------------------|--|
| 1. Look up | There is the concept of registries in WS that allows the discovery of Servents and Locators. The WS model also uniquely identifies services. |
| 2. Communications | As discussed the communication in WS is XML-based and as such is platform/implementation independent. As for security there is support for both transport-level and message-level. Using WS we can perform the required communications once the associated services have been created. |
| 3. Locator external interaction | WS provides the framework for all the external interactions for each of the components in our model. |
| 4. Locator internal operation | The internal operation of the Locator would have to be implemented with its functionality exposed via the Web Service. |
| 5. Servent external interaction | See point 3 above. |
| 6. Servent internal operation | See point 4 above. An advantage of this is that different Servents could be created depending on which underlying batch system is running on the resource. This would be transparent to the Initiator and Locator as the external interface would remain the same. |
| 7. Initiator external interaction | See point 3 above. In addition, non-XML attachments (such as JAR files) can be added to SOAP messages so the executables will be able to be transported |
| 8. Initiator internal operation | See point 4 above |
| 9. Other | The Servent, Initiator and Locator web services could be written in Java for example to ensure their portability. However, the WS paradigm ensures that they will be interoperable as long as they implement the correct portTypes for their WSDL description no matter which language they are written in. For the fault tolerance, web services provide reasonable support at the message level. |

Web Services provide the means for defining the interfaces and communication layer for our Initiator-Locator-Servent model. However, it does not provide for any of the internal functionality of the components. This is not necessarily a bad thing as it allows us to change those components at a future date or have different versions of them for different platforms, etc, whilst maintaining the interfaces. It will mean extra effort in writing the code for handling this functionality, for example job matching and monitoring.

Web Services are a well-established technology utilised in various environments in both industry and commerce. They would appear to offer a flexible solution to our model. They perhaps offer us more control over our implementations than Condor (as we are the authors!) but, as such, entail more effort in these areas. However, this would ensure that, given enough time, all the requirements would be satisfied.

3.3 EPCC ePortal

EPCC developed a prototype portal [9] allowing scientists to request job runs on remote machines (see Figure 1). Communication for the ePortal was undertaken using Java Remote Method Invocation (RMI [10]), which allows method calls to be invoked across distributed applications. We mention the ePortal here as the HPC Servent component was designed to interact with several different batch systems, such as LSF, PBS and Sun Grid Engine, so it may be useful to examine this for our Servent component.

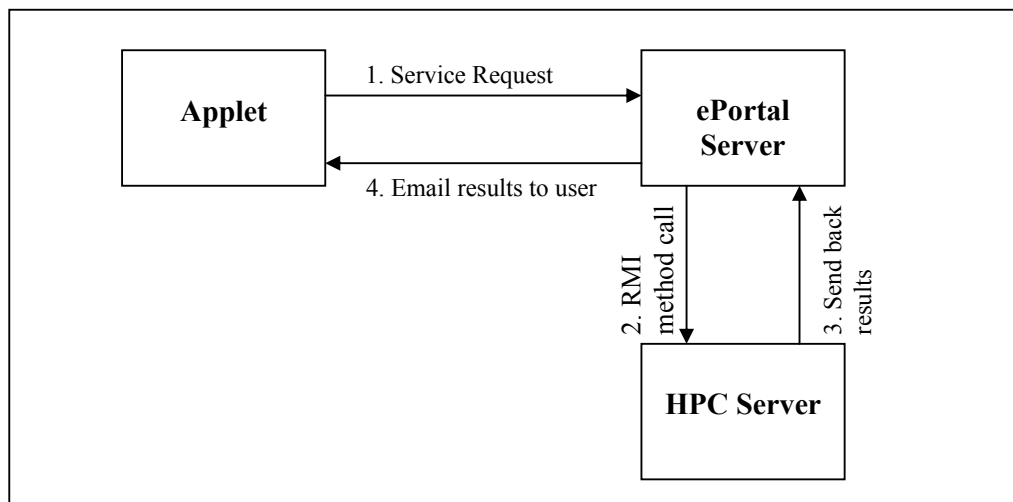


Figure 1: ePortal components

Java RMI could provide us with a communications paradigm for our model. However, it would appear that developers are currently leaning towards the Web Services model rather than RMI. I suspect this is due to the flexibility of Web Services and compatibility with those written in other languages.

3.4 Grid Services

Grid Services can be considered as Web Services with some extra features. The Open Grid Services Infrastructure (OGSI [10]) is a set of WSDL specifications defining standard interfaces, behaviours and schemas for these Grid Services. The key additional features that Grid Services provide over Web Services are:

- Transience – OGSI defines the idea of transient short-lived services. For example, in our model one could associate a service with a running job to allow querying of its progress, but one would expect this service to expire once the job has completed.

- Naming scheme – OGSI provides a two-level naming scheme, Grid Service Handles (GSH) and Grid Service References (GSR). A GSH is a type of URI or URL that is constant for the lifetime of the service and is globally unique. The GSH can be used to find the GSR, which represents the service interfaces.
- State – OGSI provides a model for accessing the internal state that a Grid Service chooses to publicly expose, providing standard mechanisms for querying, updating and adding and removing data associated with each grid service instance.
- portTypes – there are several OGSI-specific well-defined portTypes that a Grid Service must provide such as HandleResolver, Notification, etc

The OGSI specifications have been implemented in several different software platforms including Java (Globus Toolkit 3), Microsoft.NET and Perl.

3.4.1 OGSI and WSRF

From the Globus WS-Resource Framework FAQ [12]:

The WS-Resource Framework (WSRF) is a set of six web service specifications that define what is termed the WS-Resource approach to modelling and managing state in a Web services context.

The WSRF will replace OGSI. The functionality it represents is essentially the same as for OGSI but is a refactoring to take advantage of recent developments in Web Services architecture. An excellent discussion of the relationship between WSRF and OGSI can be found in [3]. The key point that should be mentioned here is that the change from OGSI to WSRF represents a change in the fundamental message exchanges and XML definitions that underlie the Globus Toolkit. Thus, while these changes are, for the most part, minor and syntactic, the effect is that a WSRF-based GT4 cannot interoperate (at the message level) with GT3. As the GT4 implementation of WSRF is not expected until Q3, 2004, this would suggest this should be avoided. As for other OGSI implementations, the effect will be similar.

3.4.2 Grid Services: Summary

The requirements are satisfied in a similar way to that for Web Services. Due to the aforementioned OGSI-WSRF transition, if it is the service-based approach we wish to take, we should consider the ramifications of choosing Grid Services over Web Services. OGSI will cease to be supported over the coming months but there is already a lot of technology using it, and the proposed alternatives will not be available until the end of our project.

3.5 GRAM

GRAM [13] is the Globus Resource Allocation Manager. GRAM provides a single standard interface for requesting and using remote system resources for the execution of jobs. Most commonly it is used for remote job submission and control. It also provides a simple authorisation mechanism for both users and remote resources based on Grid Security Infrastructure (GSI [14]) identities.

One of the ideas behind GRAM is that it provides a consistent API as a bridge between applications such as resource brokers and a platform's underlying local management mechanism. There are a broad range of metaschedulers and resource brokers that have been written to utilise GRAM, including the JOSH software developed as part of the EPCC SunDCG project (see section 3.6). GRAM does not provide scheduling or resource brokering capabilities itself. It also does not provide accounting features.

GRAM currently exists as two implementations, the GT2 and GT3 GRAM. The GT2 implementation is a set of functions written in C that provide the API for submitting,

monitoring and cancelling a job request. It requires that there is a Globus Gatekeeper running on the remote resource to accept the request. The job request is written using the Resource Specification Language (see section 3.5.1).

The GT3 GRAM provides the same functionality as GT2 but within an OGSi Grid Service-based environment. For both implementations the job request is written using the Resource Specification Language (see section 3.5.1). GRAM processes the requests for resources for remote application execution, allocates the required resources, and manages the active jobs. Usually GRAM interacts with the local batch system (for example Condor or Sun Grid Engine), but can be used as a stand-alone if no system is present.

3.5.1 RSL

The Resource Specification Language [15] provides a common language to describe resources and their attributes. Like GRAM it has two incarnations, one a text-based scripting style, the other an XML based document with associated schema. There are some standard attributes defined specifically for job submission via GRAM, the table below displays how these match up to proposed job submission criteria mentioned in [1] and discussed with Glasgow:

Table 3: RSL support for the job submission criteria

| Criteria | RSL |
|--|--|
| Number of processors (in the context of JPie, this relates to the number of individual jobs rather than multiprocessor requests) | No direct support but multiple requests on a per job basis would circumvent this |
| Minimum required memory | minMemory |
| Ideal required memory | maxMemory |
| Processor time | MaxCPUtime |
| Wallclock time (i.e. job must be completed by ...) | maxWallTime |
| Minimum bandwidth | No support |
| Maximum bandwidth | No support |

3.5.2 GRAM: Summary

Table 4: GRAM and the requirements

| Requirement | GRAM |
|-----------------------------------|--|
| 1. Look up | GRAM assumes a fully Globus-enabled environment, and as such there are other Globus Toolkit components such as the Information Services and Monitoring and Discovery Service [16] used in parallel with GRAM. These can provide registry services in the OGSi GT3 environment |
| 2. Communications | As for row 1 above, the communication is provided by the Globus environment, i.e. GridFTP and HTTP/SOAP |
| 3. Locator external interaction | The GRAM functionality allows for submission of remote jobs, but as mentioned earlier it is a middleware layer that does not provide scheduling or brokering services. It does however give us the RSL (Section 3.5.1), which covers some of the job submission criteria we require, and also provides job monitoring. |
| 4. Locator internal operation | |
| 5. Servent external interaction | |
| 6. Servent internal operation | There are various GRAM implementations in existence which interact with a variety of underlying batch systems such as Condor, LSF, Sun Grid Engine ... |
| 7. Initiator external interaction | |
| 8. Initiator internal operation | GRAM provides support for setting the runtime environment on the remote resource, and provides mechanisms for data and executable transfer between the resources |
| 9. Other | GRAM does not provide metrics for the job; it is assumed that the underlying system collects these. The Globus Toolkit has multiple platform support. |

GRAM, as part of the Globus Toolkit, provides much of the job submission functionality, and also interoperability with underlying batch systems via a single API.

3.6 JOSH

JOSH (JOB Scheduling Hierarchically [17]) is a multi-site job-management tool built on top of GT3 and Sun Grid Engine. A JOSH client allows the submission, monitoring and termination of jobs running on Grid Engines at remote compute sites. JOSH also handles the transfer of data files and executables to and from jobs. In addition JOSH employs a hierarchical scheduler to match a user's job requirements against the available Grid Engine instances.

The hierarchical scheduling tool *hiersched* runs at the client site where the user submits their job specification. Jobs are forwarded to compute sites for execution. One Grid Engine runs at each compute site. A JOSH grid service also runs at each compute site. The *hiersched* tool invokes these services to interact with the underlying Grid Engines.

Any site that provides suitable access to its files can be considered a 'data site'. The compute site where a job executes is automatically a data site since files can be read or written directly from or to its local disc. An external site may also act as a data site if it provides Internet access to files via one of the following: a GridFTP server, an anonymous FTP server, or an HTTP server (for read access only). JOSH provides the mechanisms for transferring the data and executables between these sites.

When scheduling the *hiersched* tool finds the best compute site for a particular job based on the submitted parameters. These parameters are similar to those provided by GRAM. However in addition JOSH uses *ping* to establish data transfer rates

JOSH requires a full GT3 install at the server site and access to the GT3 libraries at client sites (i.e. a service hosting container is **not** required at the client side). A resource can be both a JOSH client and a JOSH server. Currently, JOSH only operates with Grid Engine, but the software has been written in such a way that the Grid Engine-specific components have been abstracted away from the core functionality, so it should be relatively straight forward to extend the JOSH system to support other schedulers. It also should be noted that Grid Engine is available for a variety of Unix-based platforms such as Linux, not just Solaris (see [18]).

3.6.1 JOSH: Summary

Table 5: JOSH and the requirements

| Requirement | JOSH |
|-----------------------------------|--|
| 1. Look up | Currently JOSH supports a file-based registry of compute sites. Each client site has its own instance of the file that is set up 'by hand'. So the idea of a registry that can be contacted by Servents is not supported. |
| 2. Communications | Communications is handled by the underlying Globus mechanisms, and as such is also secure. |
| 3. Locator external interaction | The Locator in the JOSH system is essentially a local file with a list of resources that the <i>hiersched</i> tool reads. <i>hiersched</i> provides us with the decision-making based on the job parameters. <i>hiersched</i> currently only returns the most appropriate resource but it does rank them internally. |
| 4. Locator internal operation | Set up by hand so does not support resources registering themselves. |
| 5. Servent external interaction | The JOSH Grid Service provides the means for external interaction with the Servent, which in this case is a Grid Engine instance running on a resource. This would have to be extended to work with other local schedulers, but ideally the JOSH Grid Service portTypes etc would remain consistent. |
| 6. Servent internal operation | This is the Grid Engine instance. As such it has static and dynamic information about the resource. |
| 7. Initiator external interaction | JOSH provides some of the Initiator functionality, such as the methods for defining a job submission resource request. It also handles transfer of files (such as the JAR executable) to the remote resource. |
| 8. Initiator internal operation | JOSH does not provide the functionality for resubmitting, or submitting to other resources. |
| 9. Other | JOSH is written in Java. Some fault tolerance is provided by GT3. Grid Engine can supply statistical information on the job run. |

It would appear that JOSH would provide us with a large chunk of our required functionality. We would gain scheduling decisions, communications protocol and the reliability of established technologies in Grid Engine. Where it falls short is in this dedication to Grid Engine, but as mentioned earlier the code has been developed such that the Grid Engine specific interfaces have been separated as far as possible from other components, so it should be possible to extend to other resource managers. JOSH also does not support the Locator requirements well as this is currently a local file listing the resources rather than the registry-style service that had been envisaged. However if a Locator service was developed separately it should be reasonably straightforward to incorporate it with the JOSH software.

3.7 UNICORE

UNICORE (UNiform Interface to COmputing REsources [19]) is a selection of mainly Java-based components that allow the submission of jobs to distributed resources in a secure manner. There are five components used within UNICORE:

- Client – the user interface for submitting and querying UNICORE jobs. Each job is coded in a serialized Java object known as the Abstract Job Object (AJO) and is transmitted to the ...
- Gateway – typically running on the organisation's firewall, this authenticates job requests and forwards them to the ...
- Network Job Supervisor (NJS) – this translates the AJO into the machine specific incarnation. The resulting request is transmitted to the ...
- Target System Interface (TSI) – executes the job on the selected computing resource.
- UNICORE User Database (UUDB) – is used by the NJS to map an authentication certificate to a valid login on the computing resource.

The UNICORE Certificate Authority issues certificates to users, and security is enforced by each request being signed with this certificate and authenticated at the Gateway via a Secure Socket Layer (SSL).

The UNICORE Client is a GUI for setting up jobs, security, users and additional application-specific plug-ins. It does have a command-line interface for submitting pre-prepared jobs without having to launch the GUI. Via the GUI one can also browse and query the available resource sites. There are various job sub-tasks built in to the Client easing such activities as exporting and importing to and from the resource. The Client also allows monitoring of the jobs once they have been submitted. It is left to the user to choose which resource to submit their job to.

The Gateway only accepts connections and pass data from acceptable Clients who have presented a valid X509 certificate.

The TSIs need to be tailored to the underlying batch system running on the resource. There are examples of these provided in the documentation.

3.7.1 UNICORE summary

Table 6: UNICORE and the requirements

| Requirement | UNICORE |
|-----------------------------------|---|
| 1. Look up | The set up of the Client, Gateway, and NJS which effectively represent the Initiator-Locator-Servent relationship is performed by hand, i.e. the Client must be told where the Gateways are, each Gateway must be supplied with the related NJS and so on. In relation to our ILS model, the Client would act as the Locator, but similarly to the Condor model the Initiator-Servent interactions would have to take place via the Client as well. |
| 2. Communications | Communications is handled by the UNICORE components and is based on SSL, and as such operates across different platforms. Authentication is undertaken at each stage of interaction between the UNICORE components so it would appear they are very secure. |
| 3. Locator external interaction | As mentioned the set up of the Client must be performed by hand via the GUI. It is however a Java-based program with the source code available so it would be possible to write to its API to avoid using the GUI. |
| 4. Locator internal operation | It does maintain dynamic information resources (which can be updated on demand) but does not perform any ranking or selection against given criteria processes, these are left to the user. |
| 5. Servent external interaction | This is represented by the NJS. It does not make decisions on the running of jobs, this is left to the user at the job submission stage. |
| 6. Servent internal operation | The NJS does contain resource information such as memory, processor speed etc but does not provide bandwidth knowledge. It handles job monitoring but does not make resource decisions; instead the user makes these via the Client. |
| 7. Initiator external interaction | The Client allows the setting of job parameters excluding bandwidth. There are not resource requests as such, basically the user creates a job and then submits it via hand to the user-chosen resource. Packaging and transfer of the JAR can be handled by the UNICORE components. |
| 8. Initiator internal operation | As mentioned the creation of jobs is handled via the GUI, although the jobs themselves are well defined so it would be possible to generate them from code. There would have to be some consideration given to how to solve the lack of a ranked list of servents. |
| 9. Other | UNICORE is written in Java (apart from the TSI which are in Perl) so is portable. It is a relatively mature technology and has excellent logging facilities that would imply good fault-tolerance. The status of resources can be accessed and updated via the Client. |

The UNICORE system does supply is with much of our required functionality: there are however some omissions:

- The Client GUI – it will require some effort to write code to utilise the API for our purposes.
- User-based – UNICORE provides the mechanisms for maintaining and getting resource information but all decisions e.g. where to run, are made by the user. Again this would require considerable effort to adapt to our requirements.
- The TSIs – which interact with the underlying resource. There are no Windows examples of these so they would have to be developed.

It would appear to me that UNICORE is aimed at the application scientist who wants a straightforward way to run their jobs at different resources (which they know about), rather

than an *application* that wants automated matching to the job criteria on *any* appropriate resource.

3.8 JCSP

JCSP (Communicating and Synchronising Processes For Java [20]) is a Java API for enabling the dynamic construction of layered networks of encapsulated processes. JCSP is mentioned here as a potentially useful technology for Glasgow's $\mathcal{J}\pi$ implementation rather than directly satisfying our Initiator-Locator-Servent model. JCSP provides the Java programmer with multiple communications between processes and supports the concept of network channels, which is one of the requirements for a $\mathcal{J}\pi$ implementation. Each end of a channel has a network address, and this is mapped to a JCSP virtual channel number thus hiding the underlying communications from the programmer. However, JCSP still has the issue that only serializable objects can be passed through the network channels so it is not clear that the concept of passing channels through channels is supported. There is a channel name server (CNS) that acts as a repository for the ends of channels, i.e. read and writes similar conceptually to the $\mathcal{J}\pi$ *taps* and *funnels*. This may circumvent the necessity for passing actual channels down channels, as the CNS names may be the only information necessary for creating the connections. Certainly JCSP warrants some further investigation.

3.9 CSF

The Community Scheduler Framework (CSF [21]) is a set of Grid Services, implemented using GT3, which provide an environment for the development of metaschedulers that can dispatch jobs to resource managers such as LSF, SGE and PBS. It consists of three main services:

- Job Service – for submitting, controlling and monitoring jobs
- Reservation Service – for reserving resources on the Grid
- Queuing Service- provides a basic scheduling capability

If a job does not specify a particular machine to run on (which is the model $\mathcal{J}\pi$ uses) then the queuing service dispatches the jobs in a round-robin fashion to the next available resources.

NB: the current version of CSF is an alpha version and is only supported for x86 systems running Linux.

3.9.1 CSF Summary

CSF has the pros and cons of utilising GT3 (section 3.4.2) as its underlying technology. In addition, it provides for interaction with a number of resource managers. However additional effort would be required for implementing our own scheduler for resource selection, and implementing/testing on non-Linux platforms.

4 Recommendations

Based on the issues raised and discussed in the previous section, one recommendation would be that the JOSH software would be a good start towards our own solution to match the requirements. It has been developed by EPCC and as such there is access (albeit time-limited) to the JOSH development team. It also satisfies many of our requirements straight away and can be adapted to match the other requirements. It also satisfies the unwritten "Grid" requirement, i.e. it clearly is a true Grid technology solution and as such is in keeping with the original project proposal.

Let us be clear here though that there would still be a significant amount of effort to adapt the JOSH software to our ends. For example, within the JOSH model the Locator functionality would be split, with JOSH handling the matching against resources, and a separate look-up service, which would need to be developed, to identify those resources (see section 3.6.1). Also we would need to find an alternative to Grid Engine for running on non-Grid Engine enabled platforms (such as Windows). However, one of the underlying technologies that JOSH utilises is GRAM (section 3.5), which interfaces with various resource managers including Condor (section 3.1), which in turn has an implementation for Windows platforms. So these issues are certainly not insurmountable. For instance, GRAM itself can be used for job submission even if there is no underlying resource manager.

There is a possible concern with delays introduced by the resource managers themselves. The situation may arise where a resource seems suitable for a job but the job is submitted and ends up on a queue rather than executing. The use of the wall clock time job specification parameter should prevent this from happening. Also resource managers such as Grid Engine can usually be configured so that queuing is kept to a minimum (i.e. making itself unavailable if there are queued items, not allowing queued items etc) so in this particular case is manageable.

A possible alternative to JOSH would be a Web Services approach to our problem. I think it would allow a more iterative approach to developing the functionality, and we would be developing services specifically for the task at hand rather than trying to adapt other software to our requirements. It would also be more straightforwardly extensible to the emerging WSRF standards rather than going down the Grid Service route. However it does involve developing *everything* from scratch, and thus does not, for example, straightforwardly provide support for interaction with underlying batch systems or scheduling decision-making and so on.

Clearly some thought should be given to the platforms that we are targeting as the assumed batch systems raise some potential difficulties with any solution, including the possibility that jobs running within a batch system will not be able to launch other jobs, a key requirement of the $\mathcal{J}\pi$ system. At this point I would also comment that no matter which solution we pursue, it is important that we have a well-defined series of tests and test platforms to ensure that the solution satisfies the requirements.

5 References

1. "Use Cases: Initiator-Locator-Servent model", EPCC-IS-PGPGRID-UC-ILS
2. The Condor Project Homepage, <http://www.cs.wisc.edu/condor/>
3. "From OGSi to WSRF: Refactoring and Evolution", available from <http://www.globus.org/wsrf/>
4. MS.NETGrid, <http://www.epcc.ed.ac.uk/~ogsanet/>
5. Condor: supported platforms, http://www.cs.wisc.edu/condor/manual/v6.6/1_5Availability.html
6. SOAP specifications, <http://www.w3.org/TR/soap/>
7. W3C Note on SOAP, <http://www.w3.org/TR/2000/NOTE-SOAP-20000508/>
8. WSDL, <http://www.w3.org/TR/wsdl>
9. ePortal, http://www.epcc.ed.ac.uk/computing/research_activities/grid/eportal/
10. Java RMI, <http://java.sun.com/products/jdk/rmi/>
11. OGSi working group, <http://forge.gridforum.org/projects/ogsi-wg>
12. WS-Resource Framework: FAQ, <http://www.globus.org/wsrf/faq.asp>
13. GRAM, <http://www-unix.globus.org/developer/resource-management.html>
14. GSI, <http://www-unix.globus.org/security/>
15. RSL, http://www-fp.globus.org/gram/rsl_spec1.html

16. MDS, <http://www.globus.org/mds/>
17. JOSH, <http://gridengine.sunsource.net/project/gridengine/josh.html>
18. Grid Engine, <http://gridengine.sunsource.net/project/gridengine/download.html>
19. UNICORE, <http://www.unicore.org/>
20. JCSP, <http://www.cs.kent.ac.uk/projects/ofa/jcsp/>
21. CSF, <http://sourceforge.net/projects/gcsf/>