

12-13-2003

Service Based Marketplace for Applications

Anand Kumar Kalyanasundaram

Follow this and additional works at: <https://scholarsjunction.msstate.edu/td>

Recommended Citation

Kalyanasundaram, Anand Kumar, "Service Based Marketplace for Applications" (2003). *Theses and Dissertations*. 3927.

<https://scholarsjunction.msstate.edu/td/3927>

This Graduate Thesis - Open Access is brought to you for free and open access by the Theses and Dissertations at Scholars Junction. It has been accepted for inclusion in Theses and Dissertations by an authorized administrator of Scholars Junction. For more information, please contact scholcomm@msstate.libanswers.com.

SERVICE BASED MARKETPLACE FOR APPLICATIONS

By

Anand K Kalyanasundaram

A Thesis
Submitted to the Faculty of
Mississippi State University
in Partial Fulfillment of the Requirements
for the Degree of Master of Science
in Computer Engineering
in the Department of Electrical and Computer Engineering

Mississippi State, Mississippi

December 2003

Copyright by
Anand K Kalyanasundaram
2003

SERVICE BASED MARKETPLACE FOR APPLICATIONS

By

Anand K Kalyanasundaram

Approved:

Tomasz A. Haupt
Associate Professor of Computer
Science and Engineering
(Director of Thesis)

Anthony Skjellum
Professor of Computer and
Information Sciences, University of
Alabama at Birmingham
(Committee Member)

Yul Chu
Assistant Professor of Electrical and
Computer Engineering
(Committee Member)

Nicholas H. Younan
Professor of Electrical and Computer
Engineering
Graduate Coordinator, Department
of Electrical and Computer
Engineering

A. Wayne Bennett
Dean of the College of Engineering

Name: Anand K Kalyanasundaram

Date of Degree: December 13, 2003

Institution: Mississippi State University

Major Field: Computer Engineering

Major Professor: Tomasz A. Haupt

Title of Study: SERVICE BASED MARKETPLACE FOR APPLICATIONS

Pages in Study: 84

Candidate for Degree of Master of Science

The Grid has revolutionized the way computations are done on the Internet. Access to remote computational resources and ad hoc creation of virtual organizations across administrative domains opens new opportunities on the Grid. The newly developed web services based Open Grid Services Architecture makes the Grid more accessible by allowing the Grid to be constructed from distinct platform independent components. Together they provide an environment for application sharing (or trading), collaborations and access to remote data repositories. The application marketplace is a natural extension to this application sharing environment. The marketplace addresses the fact that the existing infrastructure is still incomplete without provisions for publishing and discovering applications and resources, including the application descriptors that must be moved between the market participants. This work demonstrates a web service instance-based infrastructure, the application market that allows the sellers, the application and the CPU providers to publish their applications for the users to find and use.

The application market uses a portal architecture built on top of Globus toolkit 3.0 that interacts with the providers and the users. The market services provide distinct interfaces that allow providers to advertise applications and users to select, configure, and run these applications. The applications themselves are modeled as stateful objects represented using XML which can be exchanged between the providers and users when required. The marketplace, through its interfaces, effectively hides the compute resource and application complexity thus allowing end users to explore and use applications unfamiliar to them with ease.

ACKNOWLEDGMENTS

This work would not have been possible without the support and assistance from a lot of people. To begin with, I would like to thank my major professor and thesis director, Dr. Tomasz Haupt for his constant guidance and support. I would like to thank my committee members, Dr. Yul Chu and Dr. Tony Skjellum for their encouragement.

I would also like to thank Dr. Puri Bangalore, Sheikh Ghafoor, Greg Henley, Nisreen Ammari and Dr. Avichal Mehra for their suggestions and help with this work. The application marketplace is built on top of the DMEFS project done by the ECS group and this acknowledgement would not be complete without a mention of the ECS team: Shravan Duruvasula, Maxim Khoutornenko, Kaustubh Kulkarni, Biju Raman, Fazal Saiyed and Raja Veeramani. Finally, I would like to thank my friends who made my stay at MSU enjoyable.

TABLE OF CONTENTS

	Page
ACKNOWLEDGMENTS.....	iii
LIST OF TABLES.....	vii
LIST OF FIGURES.....	viii
 CHAPTER	
I. INTRODUCTION.....	1
1.1 Role of grid computing.....	1
1.2 Globus and webservices.....	2
1.3 Need for an application market.....	5
1.4 Hypothesis.....	6
1.5 Organization of this document.....	7
II. LITERATURE REVIEW.....	9
2.1 An introduction to Web services.....	9
2.2 Globus services.....	12
2.2.1 Globus Resource Allocation Manager (GRAM).....	12
2.2.2 Grid File Transfer Protocol (Grid FTP).....	13
2.2.3 Meta Directory Service (MDS).....	14
2.2.4 Globus Security Infrastructure (GSI).....	14
2.3 Open Grid Service Infrastructure (OGSI) and Open Grid Services Architecture (OGSA) specifications.....	15
2.3.1 Service state data.....	15
2.3.2 Grid service instances.....	16
2.3.3 Predefined grid service port types.....	16
2.4 Globus toolkit 3.0 – a reference implementation of OGSI.....	17
2.4.1 Support for J2EE.....	17
2.5 XML tools.....	19
2.5.1 Simple API for XML (SAX).....	20
2.5.2 Document Object Model (DOM).....	20
2.5.3 Java Document Object Model (JDOM).....	20
2.6 Quality of Service.....	21
2.7 Grid Economic Services Architecture (GESA).....	21
2.8 Related work.....	22

CHAPTER	Page
2.8.1	Grid port toolkit..... 22
2.8.2	Distributed Marine Environment Forecast System (DMEFS)..... 22
III.	DESIGN REQUIREMENTS..... 24
3.1	Definition..... 24
3.2	Comparison to a conventional marketplace..... 24
3.2.1	Means of remuneration..... 24
3.2.2	Marketplace actors and their responsibilities..... 25
3.2.3	User authorization..... 26
3.3	User requirements..... 27
3.3.1	Ease of use..... 27
3.3.2	Quality of Service (QOS)..... 29
3.4	Provider requirements..... 30
3.4.1	Capturing applications..... 30
3.5	Marketplace services and application life cycle..... 31
3.6	Other requirements..... 33
IV.	IMPLEMENTATION DETAILS..... 35
4.1	Architecture Overview..... 35
4.2	Application lifecycle and the application market services..... 35
4.2.1	Introduction..... 35
4.2.2	Abstract state..... 38
4.2.3	Ready state..... 41
4.2.4	Active state..... 43
4.2.5	Ghost state..... 43
4.3	Marketplace services..... 44
4.4	The Multi-transport architecture..... 46
4.4.1	Service implementation..... 46
4.4.2	Client implementation..... 49
4.5	CPU Market services implementation..... 50
4.5.1	Database oriented services..... 50
4.5.2	The submission service..... 51
V.	RESULTS..... 55
5.1	Separation of concerns in the marketplace..... 55
5.2	Application provider's view..... 57
5.3	User view..... 59
5.4	Hypothesis validation..... 62
VI.	FUTURE WORK..... 64
	REFERENCES..... 66

APPENDIX

A	APPLICATION DESCRIPTOR XML SCHEMA.....	70
B	EXAMPLE APPLICATION DESCRIPTOR.....	80

LIST OF TABLES

TABLE	Page
1 Web service protocol stack	10
2 Application registration GUI requirements	39
3 Application description components	57

LIST OF FIGURES

FIGURE	Page
1 SOAP message skeleton	11
2 Example service data definition.....	16
3 EJB based web services	19
4 Application marketplace interaction.....	26
5 Application lifecycle.....	33
6 Application lifecycle alongside marketplace services	38
7 Metadata generation.....	40
8 Configuration GUI.....	42
9 CPU Market middleware	46
10 Multi-protocol architecture for database oriented services.....	48
11 Client multi-protocol implementation	49
12 Submission service implementation	53
13 Separate interfaces for providers and users	56
14 Application registration GUI	58
15 Application list.....	61
16 Application configuration with CPU time QOS.....	62

CHAPTER I

INTRODUCTION

1.1 Role of grid computing

The growing popularity of the Internet has changed the way computing is done. The Internet can be used to harness powerful computers from low cost desktops and portable devices. New Internet technologies enable clustering of geographically distributed resources such as supercomputers, storage systems, data sources and monitoring systems that can then be used as a unified resource and thus form what are popularly known as “Computational Grids.” The Grid envisions that anyone with access to the Internet using a simple desktop or a pocket PC has the power of supercomputers at their finger tips by utilizing the compute and data resources on the Grid. The goal is to make the Grid the computing engine of the Internet the same way the Web is the information engine. It will provide an easy to use, yet dependable and secure access to high-end compute resources, data repositories, databases, and instruments. Such an infrastructure will facilitate better use of sharable resources and tools. It will revolutionize the way software is developed, distributed, and put to operation.

Traditionally computational resources were accessed using a resource-based model wherein the users manually log into the resource of interest to run and monitor their

computation. Such a model is too tedious and inefficient as the users have to authenticate themselves every time they log into the resource. The Grid, in an attempt to solve this, uses a location-transparent services based model. In this model, the user delegates responsibilities to services provided by the Grid. In case of computational simulations, the Grid services then controls, monitors, and delivers outputs of jobs using mechanisms completely transparent to the user.

The Grid is inherently very complicated. Factors that contribute to this complication include the many network types, incompatible hardware architectures, different operating system security mechanisms and deficient protocol support in many programming languages. Hence, the solution to create such a computational Grid is understandably complicated and is a nontrivial task. Many efforts have been made to construct a homogeneous view of this heterogeneous environment. One such popular effort is Globus.

1.2 Globus and webservices

Globus is a meta-computing toolkit that defines an “abstract computing machine on which can be constructed a range of alternative infrastructures, services and applications” [1]. The toolkit addresses common issues on the Grid like communication, authentication, system information and data/resource access [1]. It is intended that the common interface provided by Globus be used to construct higher level services. Though Globus was a revolution in the way Grids were constructed, its initial implementation was, unfortunately, not perfect. The initial implementation provided distinct basic services and well defined interfaces, but the communication protocols used were still custom designed

for Globus services [2]. Simultaneously, the web services specification drafted at the W3C and promoted by IBM, Microsoft, Sun and other major companies gained popularity.

The web services are an evolution of the distributed component architecture. Conceptually, the web services are not much different from other distributed component architectures like the Object Management Group's CORBA, Microsoft's COM/DCOM or Sun's Java RMI [3]. Like any other distributed component, a web service is a collection of operations accessible over an interface using messages: It is a component of a service-oriented architecture. What makes a web service different is that it uses protocols based on the XML language. XML can describe all data in a platform independent manner: its ASCII format permits it to be freely exchanged across systems thus enabling creation of loosely-coupled applications. To make these web services possible, a whole suite of protocols to describe and interact with the services have been formulated. The best known among them are, Simple Object Access Protocol (SOAP) [4] that is used for messaging and Web Services Description Language (WSDL) [5] that is used to describe the service. These protocols are XML based and formulated by the W3C thus making them truly platform independent and non-proprietary [6].

Another aspect in which the web service scores over other distributed component architectures is in its choice of transport protocols. SOAP describes the message format but it can be delivered in any transport protocol the web service supports. The choices include HTTP, IIOP, SMTP, etc. with HTTP being the most popular choice [6]. Use of

Internet transport protocols makes the web services truly Internet friendly too. With web services gaining popularity, the Globus group, attracted by its advantages, chose to move its toolkit to the web services age. The result is the drafting of the Open Grid Services Architecture (OGSA) [7] and Open Grid Services Infrastructure (OGSI) [26] specifications which are still evolving documents. The OGSI specifies service semantics so that service interactions like errors and notifications can be standardized. OGSA, which builds on OGSI, specifies grid services, which have well defined interfaces for address discovery, dynamic service creation, lifetime management, notification and manageability: prime requirements for services on the Grid.

Though the current web service specifications are suited for most service implementations on the web, they do not address all issues the grid services wish to accomplish. For instance, grid services needed to have the concept of a service “session”, where a grid service call would base itself on a previous grid service call. The limitation arises due to the fact that the earlier WSDL specifications were designed for stateless services where service invocations were essentially independent of each other. This limitation limits usability of the current WSDL specification to specify grid services because it poses restrictions on scalability of grid services. For example, a grid service that monitors running jobs will have to respond to events from a running job in the context of the particular job. In this case, a single service needs to listen to possible multiple notifications and notify different users of the same too. The code implementation to create a service could be too complicated and new job monitoring mechanisms cannot be added on the fly. These issues were recognized by the Global Grid

Forum (GGF) [8] which, along with many companies like IBM and Microsoft, is working to create a new extended WSDL specification for OGSi within W3C.

1.3 Need for an application market

The Grid attempts to create a heterogeneous view of the resources on the Web. The user is abstracted from the interfaces to access and manage jobs on computational machines: the Grid hides platform and machine architecture complexity from the user. Though the user is hidden from computational resource interfaces, application complexity is still something that is left to the user to handle himself. Application complexity refers to the nuances in setting up an application for its execution. For example, a complex application such as the Navy Coastal Ocean Model [9] requires two parameter files, eighteen input files and more than seventy parameters (the actual numbers depend on the run conditions desired) [10]. In general, application complexity refers to tasks like setting up environment variables, location of libraries, input files, parameter files and arguments that differ from application to application: tasks that could baffle a user who is unfamiliar with the application. This work proposes an application market that seeks to hide application complexity from the user. The user interacts with the application market to obtain a convenient interface, a grid portal [11], to select, configure and run applications. The developers of the application and the computational resource providers describe the application for its use in the application market and “associate” their computational resource with the application respectively.

1.4 Hypothesis

The application market should provide an intuitive interface that lets users manage jobs and access resources while hiding the intricacies without compromising on functionality.

The application providers and computational resource providers need a mechanism to post the applications available to users. Such information about the application should capture all application information including machine specific information necessary to run the application. Such a captured application should be easily referable and accessible in the application market place. Thus, this work puts forth two hypotheses:

1. It is possible to build an application marketplace using a service based infrastructure with notification.
2. It is possible to capture a computational application in a portable format for it to be referenced in this market place.

This work proposes that a computational application can be captured in an architecture independent portable format for it to be referenced in the application market. The market, in turn, could be constructed using the instance based grid services of Globus toolkit 3.0 (A reference implementation of OGSA) [7] to create a scalable and extendable infrastructure. The important issues to be addressed by this design are related to the actual grid service instances that are created. The grid service instances are synonymous to dynamic objects that correspond to a particular class. Decisions need to be made on when the service instances are created and when they are disposed and the resources reclaimed. Other issues addressed include mechanisms to satisfy QOS requirements of

users in the grid architecture which has not been properly addressed by the current generation of grids and the security architecture of such a grid services system.

The hypotheses, if proved, would provide a new view of the computational grid that truly realizes the grid vision that every user has the power of a super computer on his finger tips. The application providers and computational resource providers have mechanisms to “advertise” their applications and resources. The application market place constructed on top of the Grid would hide application and resource complexity from the user. The user need not be familiar with the applications any more and the user now has a wider choice of applications: applications he may not even be familiar with. The application marketplace would also be completely unconstrained on the number of users and jobs it can support, be extendable when new job management mechanisms are introduced and satisfy user QOS requirements for job submission without compromising on security.

1.5 Organization of this document

The rest of the document is presented as follows:

Chapter 2 provides an overview of the Globus toolkit, a bag of services implemented by ANL/IBM that has been used to prove these hypotheses. It also describes the underlying web services technologies and tools used in this work.

Chapter 3 describes the design requirements that should be satisfied to prove the hypotheses and brief descriptions of ways to accomplish those.

Chapter 4 is a detailed implementation description of the application marketplace.

Chapter 5 validates the design and makes a decision about the acceptance or rejection of the hypothesis.

CHAPTER II

LITERATURE REVIEW

2.1 An introduction to Web services

Web services are the solution to application to application communication on the Web [12]. They are referenced using their programmatic interfaces [12]. The services are located at different locations on the Internet and higher level services could use these loosely coupled software components as black-box services to produce more value added services. Web services enable information sources to be available on the web as reusable components which can be mixed and integrated to build high level services on the web.

Fundamentally, web services are not much different from the traditional client server architectures. But, unlike current distributed component architectures like Java RMI, CORBA or DCOM that use object-model-specific protocols, web services chose to reuse Internet protocols. Using Internet protocols like Hyper Text Transfer Protocol[13] (HTTP) makes web services robust for its use on the ubiquitous Internet while at the same time making them friendly to almost all platforms and architectures. While HTTP with HTML is well suited for disseminating information on the web, it as such is not suited for machine-to-machine communication in web services. The solution is to use structured text messages (XML) [14] as parts of both the HTTP request and response. Currently, Simple Object Access Protocol (SOAP) is one of the most popular message

encoding mechanisms used with web services. Webservices.org [15] defines a protocol stack (Table 1) that web services ought to use. Traversing the protocol stack top to bottom, Service negotiation is the topmost layer, followed by workflow / discovery / registries, service description, messaging and transport [16]. Different protocols are suggested for use at these layers but the most popular ones use WSDL, SOAP and HTTP for their lower three layers.

Table 1: Web service protocol stack

Service layer	Function	Protocols
Service negotiation	Negotiate protocols used to aggregate web services, Process definition	Trading Partner agreement
Workflow, discovery, registries	Establish workflow process, discover web services	UDDI, BEPL
Service description	Describes the network service – operations supported, messages required etc.	WSDL
Messaging	Message exchange format, data encoding, routing, message level security.	SOAP
Transport	End-to-end connectivity.	HTTP, HTTPS, HTTPR, FTP, SMTP, HTTPG

If used, the SOAP request contains the name of the method and the arguments; the response contains the result of the invocation. A typical SOAP message (Figure 1) [4] is comprised of an enclosing envelope containing a mandatory body and an optional header. The optional header contains application specific information like user information and the body contains information meant for the ultimate recipient.

```
<?xml version="1.0"?>
<soap:Envelope xmlns:soap=...>
  <soap:Header>...</soap:Header>
  <soap:Body>
    <trade:GetLastTradePrice xmlns:trade="...">
      <symbol>TWX</symbol>
    </trade:GetLastTradePrice>
  </soap:Body>
</soap:Envelope>
```

Figure 1: SOAP message skeleton

A pair of SOAP messages – a request and a response, defines an operation. This operation is analogous to a method invocation in a component. A collection of these operations define an interface, a “portType” in web service terms and these web service interfaces are no different from Java or CORBA interface definitions. Currently, WSDL is the most popular standard to define a web service interface. Since WSDL is a web service interface, it is possible, for example, to generate WSDL definitions from a Java class interface that describes a Java service implementation [17]. Tools could be used to generate the interfaces as well as the SOAP messages automatically, thus, reducing the burden of the web service developer. Additionally, the toolkits could provide a hosting environment for the web services and take care of message or transport level security as well.

2.2 Globus services

The Globus toolkit is the baseline of this work. It is an open-architecture and open-source software API and services to build grid applications. The API and the services are aimed at providing support for information discovery, resource management, data management, communication, fault detection and portability without compromising security. Of all the components provided by the Globus toolkit, the most important ones for this work are the Grid Resource Allocation Manager [18] (GRAM), Grid File Transfer Protocol [19] (Grid FTP), Meta Directory Service [20] (MDS) and Grid Security Infrastructure [21] (GSI).

The initial implementation of the Globus toolkit (versions 1.0 through 2.4) was based on Globus specific protocols, which, though was widely accepted, still undermined its popularity. With the gaining popularity of web services, the Globus toolkit (version 3.0) was revamped to use web service protocols and concepts, thus improving the structure and design of these services.

2.2.1 *Globus Resource Allocation Manager (GRAM)*

GRAM is Globus's component responsible for remote application execution. It can allocate computational resources and manage submitted jobs. It can also update Resource information providers about availability of computational resources. The functionality Globus provides includes job status checking and cancellation of jobs. The most useful functionality for this project provided by GRAM is updating of job status using an event driven push model. In this model, the client can register a listener with GRAM to listen to job status changes. GRAM then notifies the listener of these events. The architecture of GRAM has undergone major changes from version 2.2 to 3.0. GRAM 3.0 features:

- An XML based Resource Specification Language (RSL) for job resource specification
- A WSDL interface to access GRAM
- A special user hosting environment to manage user jobs that run using the user credentials.

2.2.2 *Grid File Transfer Protocol (Grid FTP)*

Access to distributed data is an important requirement on the Grid. Scientific and engineering applications typically read large data sets and create new data sets. Grid FTP is Globus's solution to accommodate all data storage and access models on the Grid. To be precise, Grid FTP provides a high performance, secure robust data transfer mechanism that is based on FTP. It aims at providing a common data transfer protocol for all customized data storage systems like DPSS, HPSS, DFS and SRB and avoiding customized clients for specific storage systems [22].

The current implementation provides the following features:

- GSI security (user authentication and authorization based on GSI certificates)
- Parallel data transfer using multiple TCP streams
- Data transfer using third party control
- Support for reliable file transfer (restarting failed transfers, fault recovery etc.)

2.2.3 *Meta Directory Service (MDS)*

MDS is Globus's grid information service, which is critical to operation of the grid. It is designed to provide scalable access to dynamic data, support multiple information sources and allow uniform access to information. MDS makes resource information available from LDAP or other directory protocols the resource could support. The initial implementation of MDS was based on a central organization server into which the resources "pushed" updated information. This implementation, understandably, does not scale well. The MDS-2 architecture makes resource information available using Grid Resource Information Service (GRIS) [23] servers that run on the resource or Grid Index Information Service (GIIS) [24] servers that provide collective information about cooperating resources [20]. The latest implementation of MDS (OGSA based) is based on web service factory architecture. The factory spawns an information provider at the user's request [25].

2.2.4 *Globus Security Infrastructure (GSI)*

GSI is a mechanism that is built into Globus services for authentication and secure communication over the network. It additionally provides single sign-on, mutual authentication and delegation which are useful on the Grid. To provide these security services on the Grid, GSI uses X.509 certificates and the Secure Sockets Layer (SSL) communication protocol. The implementation features:

- Certificates for all users and services on the Grid for authentication.
- A channel for secure communication based on symmetric keys (established after authentication) if desired.

- Default message integrity using signed message digests.

2.3 Open Grid Service Infrastructure (OGSI) and Open Grid Services Architecture (OGSA) specifications

The OGSI specification, in short, defines a distributed component model that extends the current web service specifications, especially, the WSDL and XML schema specifications [26]. Its purpose is to introduce the concept of stateful web services, web service portType extension, asynchronous notification of state change and service state data. The new extended specification, popularly called GWSDL [27], is currently influencing the WSDL 1.2 specification. The specification also specifies a base set of common interfaces that grid services can implement. The Open Grid Services Architecture (OGSA) builds on OGSI to integrate grid technologies with OGSI-modified web services.

2.3.1 Service state data

The service state data represents the state of a stateful web service. When compared to object-oriented programming, the service data roughly parallels object attributes, which are specified as a part of class definition. Hence, OGSI specifies that the service state data definition too should be externally observable along with the service definition. The service data definitions are added along with port type definitions as shown in

Figure 2. The service data distinguishes one grid service instance from another [26].

```

<gwsdl:portType name="someService">
  <wsdl:operation ...> </wsdl:operation>
  <sd:serviceData name="jobStatus" type="xsd:String"/>
</gwsdl:portType>

```

Figure 2: Example service data definition

2.3.2 *Grid service instances*

Instances of the same grid service are described by a single grid service description, but differ in their service data content. Going back to the object-oriented programming analogy, grid service descriptions parallel class definitions and grid service instances compare to objects. Grid service instances are referred to using one or more grid service handles (GSH). A GSH is just an instance name in the form of a URI. For it to be of any use to the client, the client should resolve the GSH into a Grid Service Reference (GSR), which describes the instance (the grid service description).

2.3.3 *Predefined grid service port types*

OGSI has identified common functionality that would be required by many services and provides these as port types to ease the burden on the developer. The developer just needs to extend these port types to obtain desired functionality. Notable port types include:

- **GridService portType:** All grid services extend this port type. It provides operations to find, query, set and delete instance service data.
- **HandleResolver portType:** Resolves a GSH into a GSR.

- NotificationSource portType: Allows grid service instances to send notifications. It provides operations to manage clients that subscribe to these notifications.
- NotificationSink portType: Provides operations to receive notifications from notification sources.
- Factory portType: This port type spawns grid services. It defines the *createService* operation to create grid service instances.

2.4 Globus toolkit 3.0 – a reference implementation of OGSi

One of the major motivations for the Globus group to move to this new architecture was that the earlier implementation had all services isolated: development of one service rarely contributed to another. The new OGSi based toolkit provides a framework for building and deploying services that makes development of grid services straight forward. The toolkit, in addition to all the standard Globus services, provides tools to generate GWSDL interfaces and web service stubs using modified Apache AXIS tools. The toolkit also supports an Apache Tomcat [28] or Microsoft .NET [29] based web server to serve as the service hosting environment. It could optionally interface with IBM Websphere [30] as well as Jboss [31] EJB servers. This toolkit is also referred to as the OGSA toolkit.

2.4.1 Support for J2EE

One of the interesting features of the toolkit is its ability to expose Enterprise Java Beans [32] (EJB) hosted in an EJB container as a web service. The EJB container managed persistence architecture simplifies coding of database oriented services – services that

largely interact with a database. Writing the grid services as EJBs also lets the user take advantage of the features offered by the reasonably mature J2EE technology; describing instance service data in this case is still an unresolved issue though.

Web services based application market services can be created if an intermediate component that can convert web service invocations into Java RMI is used on top of J2EE [32]. Tools provided with the toolkit generate web service “redirection” stubs from the EJBs. The Globus toolkit stubs (one stub per EJB service) that are hosted in the web service container relay web service invocations from the users as RMI calls to the EJB container.

In a typical usage scenario (Figure 3) the client initially creates a remote interface, an instance of the stateless service (if one does not already exist). Once an instance of the service is created, the client makes its service invocations and the OGSA service stub hosted in the web service container receives the user request. The stub converts this invocation into an RMI call and forwards this request to the EJB container that hosts the business logic session beans. The session beans that provide data oriented services make use of the data objects that are entity EJBs to access the database to process the user request. The result of the invocation is sent back to the client through the OGSA stubs.

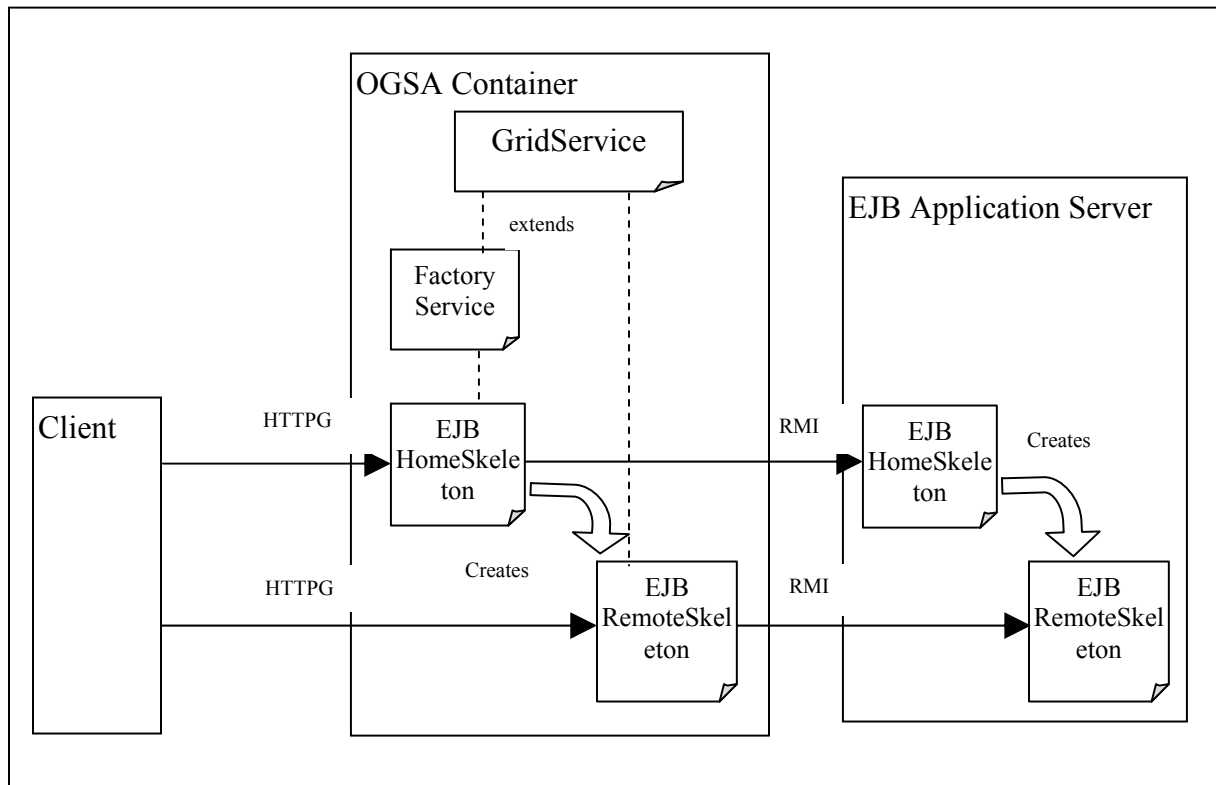


Figure 3: EJB based web services

It is worth noting that the EJBHomeSkeleton itself implements the OGSA factory specification, thus permitting it to create remote skeleton service instances.

2.5 XML tools

XML is a self-describing data format. Its ability to encode rich data formats enables it to be used for data transfers between dissimilar systems [33]. This ubiquitous data format can be produced by and used in all languages and databases including legacy COBOL systems, which is the major motivation for its use in this work [33]. Since the induction of XML into computer science, many technologies have been developed to use XML from programs: the major technologies include SAX, DOM, XSLT and JDOM [34].

2.5.1 Simple API for XML (SAX)

SAX is an API to work with XML. It is designed to handle large XML files without being a memory hogger and is well suited to performance sensitive code. SAX's approach to XML parsing is event based. It generates an event for every feature found in the XML document being parsed. Thus the program operates by responding to events based on the XML data [33].

2.5.2 Document Object Model (DOM)

The W3C Document Object Model is a "platform- and language-neutral interface that will allow programs and scripts to dynamically access and update the content, structure and style of documents. The document can be further processed and the results of that processing can be incorporated back into the presented page." It defines a programmatic interface for XML manipulation. It is an object-model based API in which the DOM parsers create an in-memory object model of the XML document. The memory now contains a tree of the DOM object that represents the structure and content of XML. DOM is a feature rich and powerful API, but its in-memory representation could become a memory hogger for large XML documents [33].

2.5.3 Java Document Object Model (JDOM)

The JDOM is an API that is tuned towards XML manipulation from within Java. It builds on top of SAX and DOM and is a more convenient replacement for DOM to build an XML document [35, 34]. The JDOM API gained popularity a considerable time after this work began and consequently has not been used in this work.

2.6 Quality of Service

Quality of service using Resource reservation is an area of research on its own and is expected to be incorporated into emerging distributed services. Various resource reservation mechanisms are available for different types of resources. For example,

- CPU reservation using DSRT [36], Start Time Fair Queuing [37]
- Network bandwidth using RSVP [38]
- Disk IO bandwidth scheduling using Cello [39]

While tools based on these mechanisms directly control the resource, they do not provide a convenient interface to be actuated from the Grid. GARA [40] is an architecture for advanced reservations that addresses this issue. It provides a convenient API to reserve resources on the Grid. Unfortunately GARA's supported list of resources that can be reserved is still preliminary. An alternative work around is to use GRAM with native resource reservation mechanisms and that is the approach used in this work.

2.7 Grid Economic Services Architecture (GESA)

GESA [41] is a part of GGF that aims at defining protocols and service interfaces to charge for OGSA based grid services usage. The goal is to create an infrastructure to facilitate organizations to be financially compensated for providing resources. GESA services defined for this infrastructure will add new service data elements and extend the OGSA specifications but are not allowed to change it. Two new services are expected to be defined by GESA – Grid Banking service (GBS) (to record financial transactions) and Chargeable Grid Service (CGS). Of these, CGS, which is most relevant to this work, extends the Grid Service port type defined by OGSA [42]. Additional operations and

service data elements allow the CGS to negotiate transaction mechanisms, define acceptable GBSs to validate and implement transactions etc. The GESA specification is a work in progress and is not complete as of this writing.

2.8 Related work

2.8.1 Grid port toolkit

The gridport toolkit [44] is a portal (A single comprehensive interface to access multiple services on the web [43]) based on the older Globus toolkit to access computational resources. It started as means to construct a web based interface to provide resource status information and a way to access HPC accounts at remote resources; it was later expanded to take full advantage of the features offered by the Globus toolkit. It currently supports five functions – management of user accounts and portal space, user authentication based on certificate repositories, job submission using Globus GRAM, simple command execution and file transfer between compute resources and portal user file space [44]. The toolkit was implemented using Perl/CGI and was designed to be accessible using a simple web browser (a browser that does not support client-side XML processing or applets).

2.8.2 Distributed Marine Environment Forecast System (DMEFS)

DMEFS [45] is a research project to develop and remotely access climate, weather and ocean models. The goal was to construct a collaborative environment that permits diverse users (model developers, operational users and portal administrators) to develop, share and validate computational models, thus, resulting in faster times to transition a model into operational use from development. The DMEFS project, which was based on

Enterprise Computational Services (ECS) [45], was designed to abstract a common user from model intricacies by using the application metadata [45] to describe models and to permit sharing of model data. The DMEFS supported two interfaces – a web browser based client that was developed using Java servlets and a Java swing based GUI client that supported a multi-protocol architecture (Section 4.4). The author was originally a part of the DMEFS development team and made significant contributions to metadata processing, model configuration, web based submission and the multi-transport architecture. This work reuses the Java swing based front end that was developed for the DMEFS project.

The DMEFS project, though a significant effort in the field of grid computing, suffered from the limitations of WSDL 1.1 (implemented by Wasp 4.0 [46] web services toolkit) that was used to build the services: the submission service was static and a single instance had to manage all user job requests. A second attempt was made to produce a better implementation by using a web service factory [17]. The submission service factory could now spawn service instances that managed user jobs, but the implementation still lacked support for service instance lifetime management and job status notifications.

CHAPTER III

DESIGN REQUIREMENTS

3.1 Definition

The application marketplace is defined as an environment where the providers and the customers interact. The providers are the class of marketplace users who “sell” applications and computational power. The customers are the end users of the marketplace who utilize and “pay” for applications and the CPU they use. It is envisioned that different classes of users will see a different facet of the marketplace: The providers should see an interface that allows them to advertise application and resource information and the users require interfaces to browse and utilize these applications and resources.

3.2 Comparison to a conventional marketplace

The marketplace for applications, in many ways, is similar to the conventional market. It embodies the two important aspects of any market, namely,

1. It has a means of payment in some form.
2. It has actors who interact with the market – the providers and the users

3.2.1 Means of remuneration

A means of payment is essential in the application marketplace: the providers expect to be compensated for the services they provide. In this case, the payment may be in various

forms: It could be in terms of allocated hours on a computational resource or CPU leased to be paid for time used. Negotiation of payments and its implementation in the Grid is the subject of research of the GESA (Section 2.7) group. Attempting to redefine the requirements and architecture for payments is considered beyond the scope of this work. The specification of GESA, when complete, can be used to create chargeable grid services for the application marketplace.

3.2.2 Marketplace actors and their responsibilities

The application marketplace actors are composed of the sellers, the application and CPU (or computational resource) providers and the buyers, the application users. The providers – application and CPU providers publish their applications in the application market. To be precise, the application providers code the application on target machine architecture(s). They collaborate with the CPU providers to install their application on the computational resource. The application and CPU providers are together responsible for publishing this application along with CPU specific options in the application market. This process is called application registration. The users browse, select and run the applications of their interest (Figure 4).

The CPU providers own the compute resource on which the application is installed; they are the administrators. Once the application is installed on a compute resource and the application is registered in the application market, the users directly interact with the CPU provider to submit their jobs. Though, in general, there may be three entities – the application developers, the CPU providers and the users, in practice, the application and

CPU providers could be the same and hence this work considers the difference between the two categories subtle and insignificant for the purposes of this work. The rest of this document would refer to both categories as just an application provider and the term “CPU provider” would be used only when it is necessary to emphasize the owner of the resource.

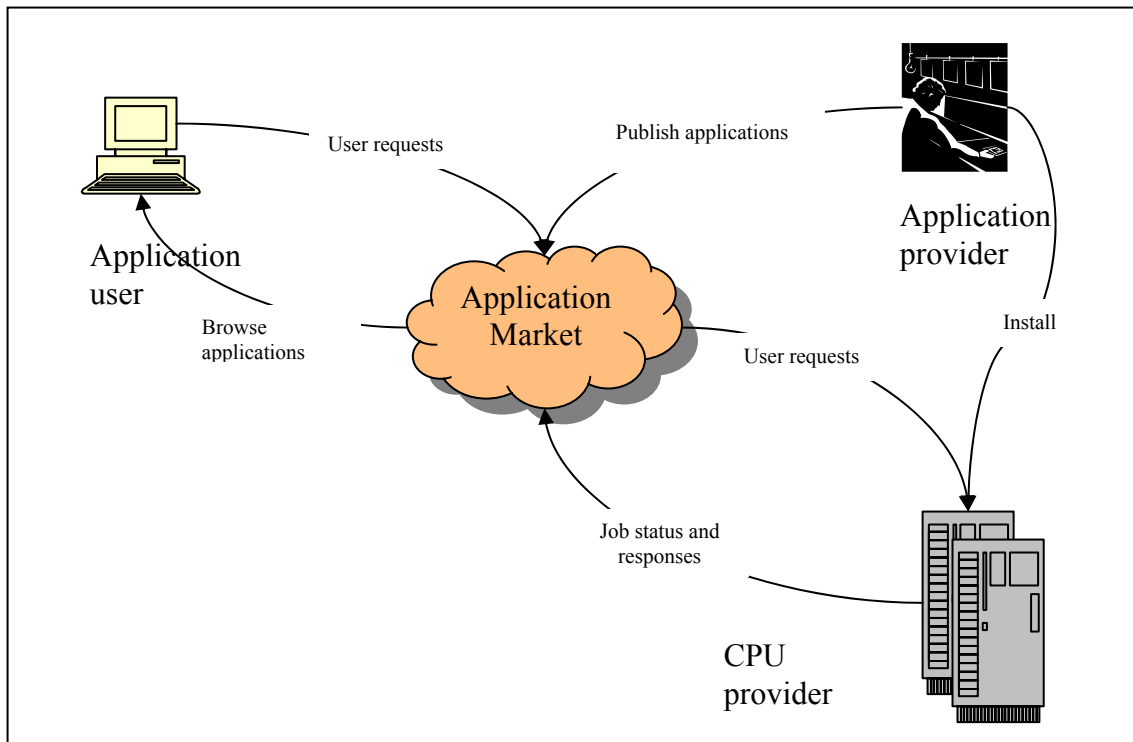


Figure 4: Application marketplace interaction

3.2.3 User authorization

Similarities apart, there are some stark differences between an application market and an ordinary market that makes its realization non trivial. One such difference is that the computational resources are partitioned between different administrative domains. The administrators control access to their resources; they are the gate keepers. Hence, each

user who wants to consume CPU time in the application market can only do so if he/she is authorized to access the resource. Consequently, the application market is responsible for establishing the identity of each user to the resource administrator. It should be noted that application market with the help of the underlying Grid can prove the authenticity of the user, but the CPU provider decides what the user is authorized to do.

3.3 User requirements

3.3.1 Ease of use

The user should be shielded from inherent characteristics of this distributed application marketplace like authentication mechanisms, machine heterogeneity and application complexity. Multiple authentication mechanisms and machine heterogeneity stem from the multifarious hardware architectures and platforms the CPU providers could choose to associate with the marketplace. Understandably, these are some of the issues addressed by the computational grids [47]. Different research groups have adopted various solutions to provide uniform authentication and access mechanisms to geographically distributed computational resources; solutions ranging from a web portal architecture used by websubmit [48] to the bag of services architecture developed by the Globus [49] group. Hence, resource authorization and access could be easily handled if the application marketplace were implemented as a services-based three-tier portal architecture (a single entry point to access multiple resources) operating on top of the grid. Such an architecture would allow:

- Single point sign-on for users.

- Easy to use interface to access applications on geographically distributed machines.
- Secure data transfer to and from applications for file stage-in and stage-outs.

Additionally, the portal architecture could also be used to handle application complexity.

To better understand application complexity, it is necessary to recognize the processes involved in running an application. Before running an application, the application is typically setup by creating or transferring the input files and setting up the parameter files. This process could be tedious and/or baffling to the user depending on the expertise of the user and his acquaintance with the application. To make this process an ease for any user, the application market should provide help with configuring an application. The application market should offer descriptions of the various application configuration components in a language understandable to the user. Specifically, it should provide uniform interfaces to specify:

- input file locations,
- parameter files and parameters,
- and application arguments.

Additionally, it should automatically transfer input files as required by the application from locations configured by the user.

The next step is job submission followed by job monitoring. These processes are so tightly coupled to the machine architecture that it is of real value to provide a uniform

user interface to perform these functions in an application market. Specifically, the application market should:

- Manage all chores related to job submission: It should be able to gather command line arguments, setup the application execution environment, create batch submission scripts if necessary and submit the job.
- Notify user regarding job status changes.
- Reserve resources according to QOS requirements.
- Handle the IO produced by the application.

3.3.2 *Quality of Service (QOS)*

In general, QOS requirements could be specified for any resource including CPU, network bandwidth and disk activity bandwidth. But CPU QOS requirement is most relevant to the application market and hence will be the focus of this work. Such QOS user requirements address the issue of value that an application result poses to the user. Computational CPU users who “pay” for their CPU time cannot accept an indefinite waiting time for their job to start. There are some approximate and worst case algorithms and mechanisms proposed to calculate the waiting time of a job in a queue. Unfortunately, such algorithms only provide an estimate of the start time of the job and do not guarantee the actual start time. While such estimation would suffice on a computational grid, the application market should provide more than an estimate. An application market should be able to guarantee the job start time and should keep the CPU(s) reserved for the time period it would take for the job to complete.

Understandably, the only solution available at this time to guarantee such a QOS is CPU reservation. Hence, an application market should be able to publish a list of the current reservations available and make reservations on computational resources as requested by the user. In general, the QOS requirements should allow the user to specify

- The architecture dependent memory requirements of the application – the minimum and maximum memory requirements.
- The CPU requirements of the application – the minimum and maximum wall clock times.
- The last acceptable start or end time of the application.
- The Bandwidth requirements of the application.

Hence, the design of this system would be to have an extendable, scalable QOS oriented application market grid computing system that can reserve resources on computational machines as required by the user (if such an allocation is possible).

3.4 Provider requirements

3.4.1 Capturing applications

While most of the requirements of the application market are defined from the user's perspective, the application providers also need a mechanism to publish their applications in the application market. Each application in the application market needs to be “captured” so that it can be referenced and accessed by the market users. Haupt [50] has

identified the following aspects of an application that need to be registered by the application developer to capture the application:

- Name and description of the application.
- Syntax, order and value of command line arguments.
- Location and names of parameter files, the parameter names and corresponding values.
- Location and names of input and output files
- Architecture independent and architecture dependent QOS requirements.
- List of machines on which the application is installed.
- Access mechanisms and batch systems (if any) installed on those machines.
- Location of the executables, input and output files for each machine.
- The user's scratch working directory for an application.

Since each user could have a different hardware and operating system to interact with the marketplace, the “captured application” should be expressed in a portable format that is understood by all user platforms. Such a “captured application” would allow all users to configure, locate and run the application.

3.5 Marketplace services and application life cycle

The marketplace services outline the interfaces through which the providers and the users interact with the marketplace and the application lifecycle along with the user requirements define these services. The application lifecycle refers to the string of events starting with the induction of an application into the marketplace through its consumption by the users to its finale with the archival of results produced by the application job run.

Clearly, the actions of the marketplace actors move the application through its lifecycle and the marketplace services provide the means for these actors to move the application through its lifecycle. Thus, each application can be treated as an object that is acted upon by the services (on behalf of the actors). To better understand the application object and what these marketplace services should be, the application lifecycle has been divided into four stages [17]:

1. Abstract state: The state when the application is installed on the backend machine. The executable has been put in place and is ready to go as soon as the execution environment has been created. The application, at this state, can run with just default information.
2. Ready state: The application reaches this state when all configuration files have been created and the input files are in place. The application just needs to be started. It should be noted that one abstract state application can create multiple ready state applications: it is a one to many relation.
3. Active state: A configured application (from ready state) enters active state when it is submitted for a run. Here too, there is a one-to-many relationship between the ready and active state applications. The active state applications have runtime information appended like application start time, batch submission queue name etc.
4. Ghost state: Once a job is complete, it enters the ghost state. At this state, the output files and configuration are captured and archived for future reference. Each active application produces exactly one ghost application.

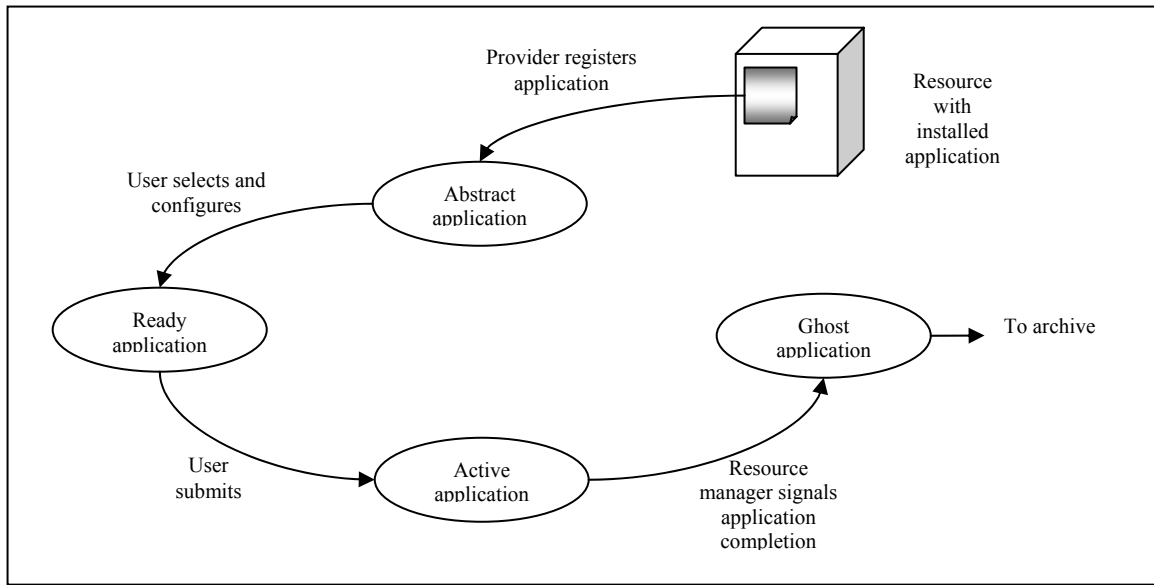


Figure 5: Application lifecycle

The providers start the application at its first stage (Figure 5), the abstract state, which marks the initiation of the application into the marketplace. When the application is selected and configured by a user, it moves to the ready state. Subsequent submission of the application (by the user) causes the application to become active. Finally, the application becomes a ghost when it completes execution.

3.6 Other requirements

It is essential that these marketplace services be accessible to a large population of users. Users could prefer some architecture based native protocol implementations that are faster or considered more secure by the client. While the need for machine architecture independence is an important consideration for selecting a Remote Procedure Call (RPC) based mechanism, such specific protocol requests should also be honored. Thus, it is imperative that provisions be made in the application market for services to be made

available through multiple RPC protocols and the client be allowed to use a protocol of its choice.

Additionally, the application market should be extendable and scalable. CPU providers should be able to join the application market when they desire. This also means that many unknown machines and architectures may need to be supported as the application market grows. Support for new mechanisms should be introduced without requiring that the entire application market be shutdown. A scalable architecture also requires that resource selection and job management chores continue to function seamlessly even when the number of users increases. In short, the application market should be able to grow without affecting current market place activity.

CHAPTER IV

IMPLEMENTATION DETAILS

4.1 Architecture Overview

For the marketplace implementation, this work follows the suggestions made in section 3.3.1. Implementing the marketplace using a portal architecture (three tier architecture) satisfies the marketplace requirements. The marketplace is hosted as a group of services by a third entity that interacts with both the application providers and the users. As a result, the client and backend components interact through a middle tier responsible for business logic and process management. The marketplace services that function on top of a Globus grid cater to the needs of both the application providers and the users.

4.2 Application lifecycle and the application market services

4.2.1 Introduction

The operations required in the marketplace are closely coupled to the application lifecycle (Section 3.5). As a result, the implementation will prepare the functions required in the marketplace and then group the same as services later in the implementation. To transition the application lifecycle into an implementation, the application itself can be formulated as a stateful object that transitions from one state to another during its

lifecycle. Such a stateful application object is too complex to be expressed as simple state variables. It also needs to be passed to the marketplace actors when required. Hence, this work uses the XML based application object definition developed for the DMEFS project (Section 2.8.2). The XML based object definition document can be represented using a single state variable and it also keeps the object definition decoupled from the service implementation.

To elaborate, XML is a markup language, thus its hierarchical structure can be used to store marked up categorized application information in this work. Such marked up XML content makes the information captured about the application self explanatory. The captured application, also called metadata (an XML document), divides data collected into many categories. Each category is marked by an XML tag and sub categories are marked by sub-tags. For the application XML object, the following categories are captured:

1. The application signature. Information captured in this category includes the name, keywords, version and authors of the application.
2. Description and documentation information.
3. Registration information: Information about who registered it, when it was registered and when it was last modified.
4. Command line arguments: Each argument to the application is captured for its syntax and order.
5. Parameter files: Each parameter file forms a sub category. The parameter files themselves have the file parameters as their sub category.

6. Input files: Encompasses the symbolic file name, a description about the file and description of the file format.
7. Output files:
8. Custom GUI
9. QOS information
10. Information specific to a target host which includes the runtime information for the application.

Such a captured application object (Example shown in Appendix II) grows in its information content as it proceeds in its lifecycle. In its initial stage, abstract state, the application object contains all default values and information necessary to customize them. The next stage, ready state, represents a customized application object suited to the user needs; it includes runtime information necessary including batch queue names, actual locations of input files and run specific parameter values. The active state application is an application that is currently being run. Finally, when the application run is complete, the ghost state application object is a “record” of the run time configuration in addition to locations of the application outputs.

Correspondingly, at each application state, the marketplace provides a different set of services for the marketplace actors to use. The provider using his interface (Figure 6) creates the initial abstract application object. The other interfaces allow the end user to transition the application to active, ready and ghost states. The accessories and operations required at each state of the application are discussed in the following sections.

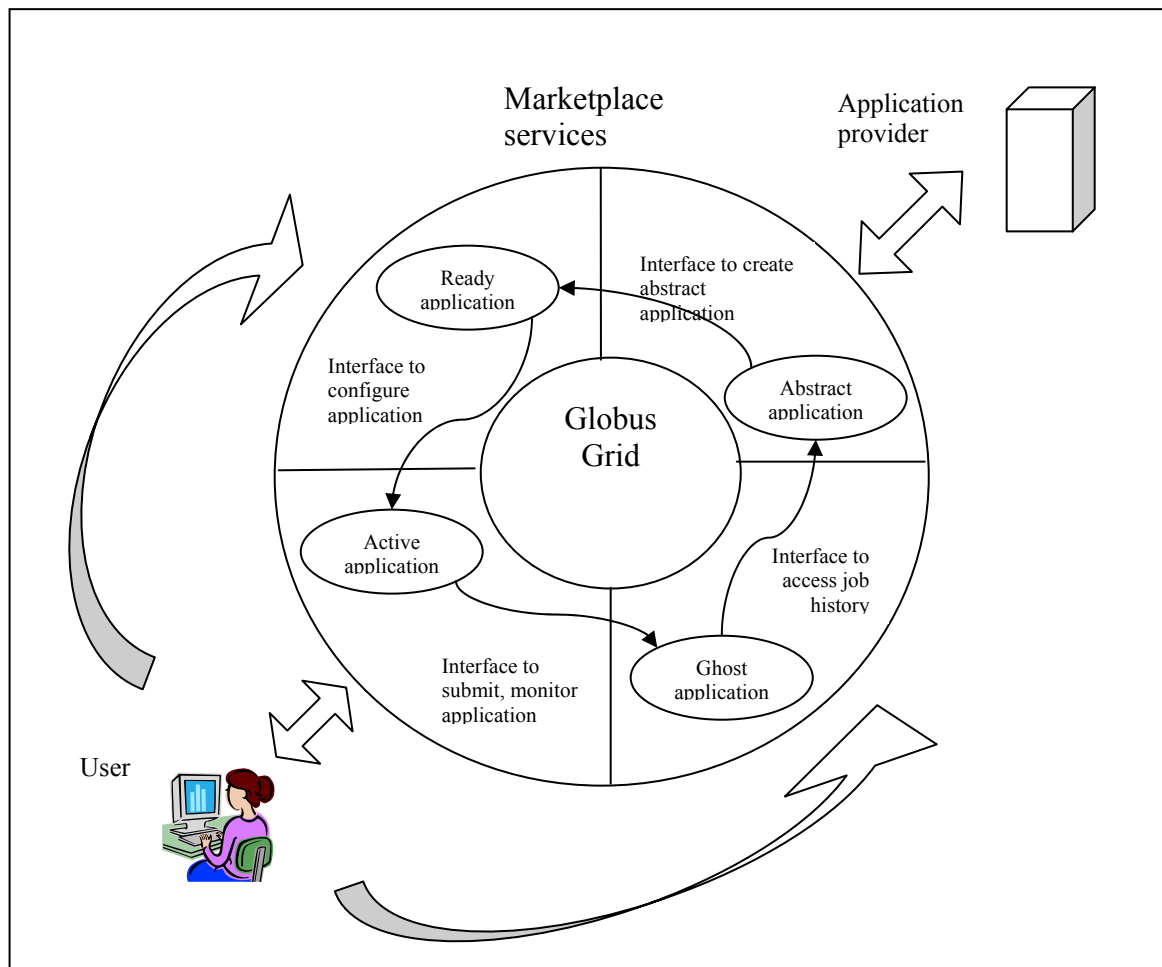


Figure 6: Application lifecycle alongside marketplace services

4.2.2 Abstract state

The provider creates the initial XML document that represents the abstract application. Writing an XML document by hand is both error-prone and tedious. Hence, this work

provides a GUI for the developer to use to create this XML metadata. The requirements for this GUI include:

Table 2: Application registration GUI requirements

- | |
|--|
| <ol style="list-style-type: none"> 1. A very intuitive interface that the developer can understand and use without any knowledge of the underlying XML format. 2. Support for extension of the XML metadata schema without requiring any change to the GUI code. 3. Generic support for XML generation from different GUI input mechanisms. For example, the GUI could be a HTML form filled out using the web browser or a Java Swing application. |
|--|

To fulfill these requirements, the XML generation module should be reusable and decoupled from any GUI: separating processing from presentation. Consequently, this process is divided into two parts – the GUI module and the XML generator module. The GUI module converts the information entered by the developer as a series of name value pairs and delivers it to the XML generator. The XML generator interprets these name value pairs in the context of the XML schema and generates the XML metadata.

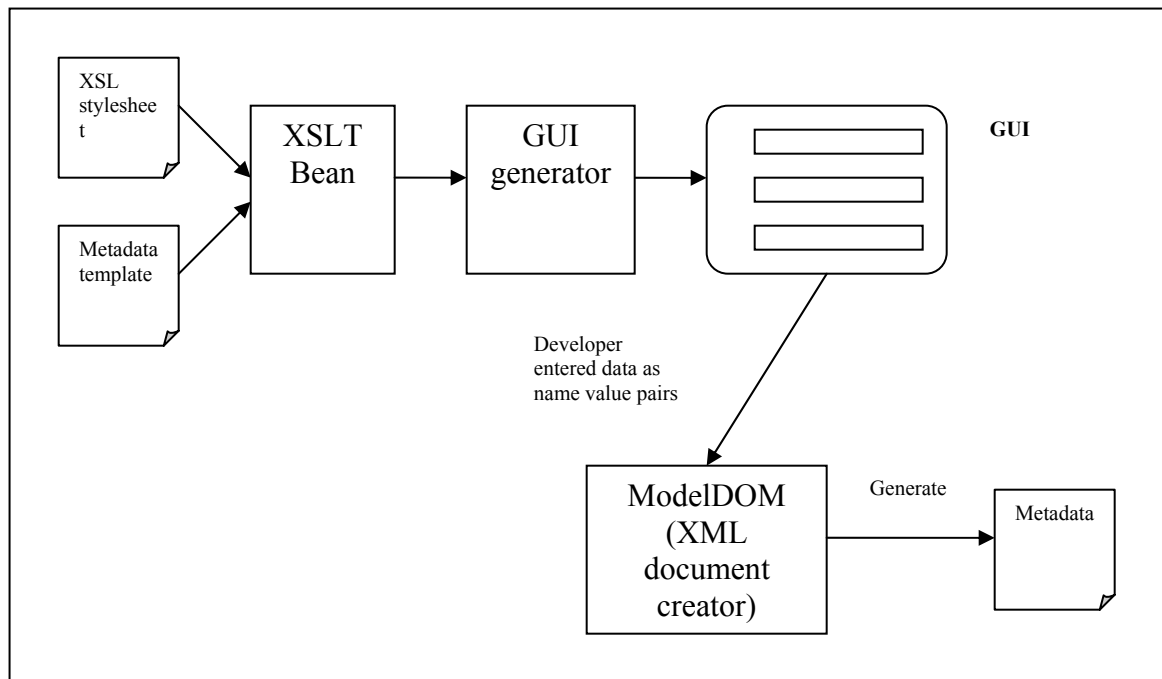


Figure 7:Metadata generation

For XML processing, the XML technologies, DOM and XSLT were used. XSLT is used to generate the GUI from the skeleton thus requiring a different technology dependent XSL stylesheet for each GUI mechanism supported. XML DOM processing is used to create the XML document from the name value pairs returned by the GUI.

The XML document, thus generated, is introduced into the marketplace. Application providers use the *newModel* operation to add new applications to the marketplace database. Users, on the other hand, use *getModelList* and *getModelInfo* operations to get a list of applications and get more information about a particular application respectively.

4.2.3 *Ready state*

An abstract application contains information that describes configurable options of the application. Specifically, such configurable options include

- Command line arguments (order, syntax, datatype and default value describe them)
- Parameters in parameter files (datatype and default value describe them)
- Values of environment variables
- Actual location of input and output files.

Configurable options need to be tuned for each run of the application before the application can be executed. Thus each abstract application needs to be configured before it becomes “ready” for submission.

Application configuration is done by users who are more interested in the outcome of the application and are typically not the application developers themselves. Hence, to make the application configuration more understandable to people who are not very familiar with the application themselves, the configuration process is presented using an application configuration wizard (

Figure 8).

The software design requirements for creating the ready application are not much different from creating the abstract application XML document metadata in the first place

(Table 2). Hence, the software design of the configuration GUI (to create the ready application) follows the same software patterns used to create the abstract application (Figure 7). The only difference being that the GUI is created from the abstract application metadata (thus filling the GUI with default values) rather than a template metadata. Once, all the information about the application run has been captured, and the ready application created, the application configuration is complete.

At this stage, all information necessary for a particular run have been captured. Such a configured application is now termed as the “ready application,” is ready to be submitted.

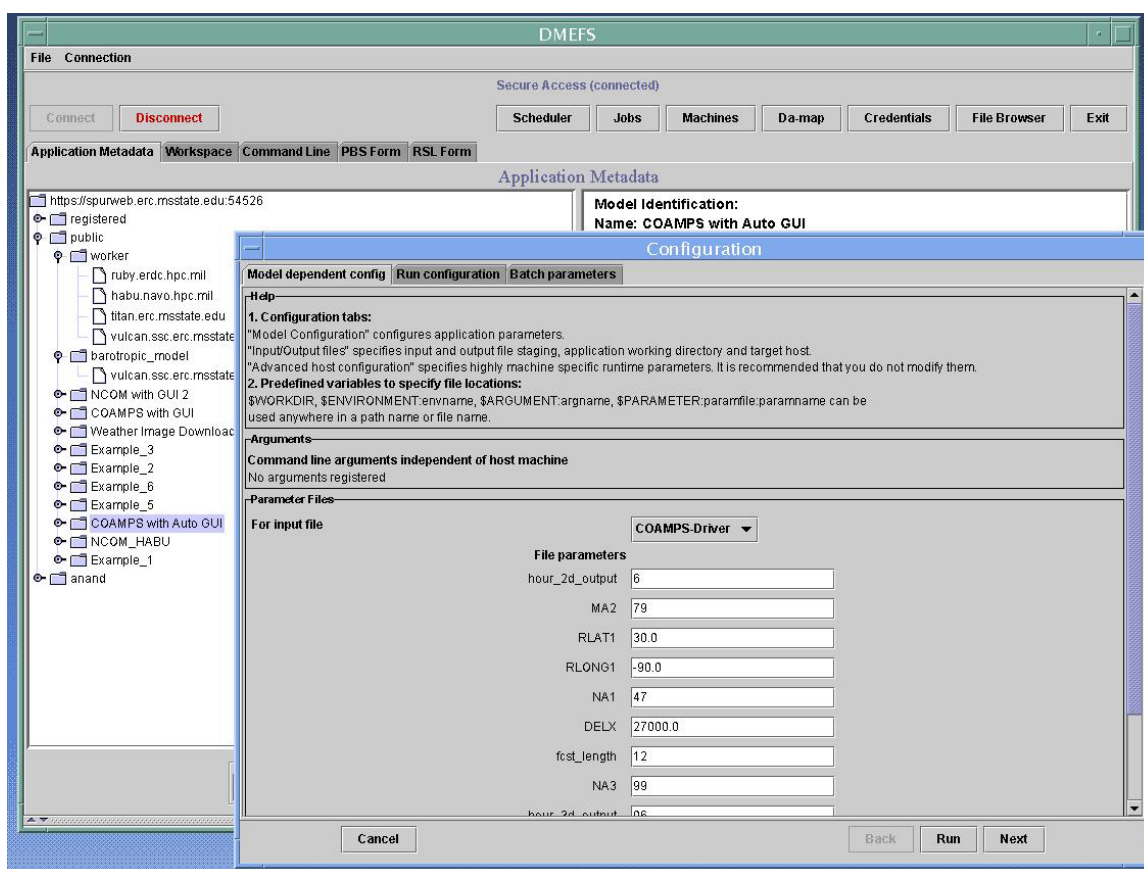


Figure 8: Configuration GUI

At this state, the user could optionally save the application in his workspace using the *newApplication* operation.

4.2.4 *Active state*

An application becomes active when it starts executing. The operation, *submit*, is used in the marketplace to submit a ready application. The job submission service is the heart of a computational grid. Its responsibilities include support for job monitoring and control, automatic job status update and if necessary, job output and error stream forwarding. The input to the *submit* operation for job submission service is the metadata proxy in its ready state with all configuration necessary to run the application. The job submission service, then, submits such an application to the job queue taking care of service QOS requirements, if necessary. The proxy is now “Active”. The current status of the job and the job properties can be obtained on demand by retrieving the submit service service-data, *SubmitData*. The user could optionally subscribe to job status change notifications by subscribing to the *SubmitUpdate* notification topic provided by the submit service. The user could also cancel his running job by invoking the *cancel* operation of the submit service. Section [4.5.2] discusses the submission service implementation.

4.2.5 *Ghost state*

The running application becomes a “ghost” when it completes execution. The resources that were used by the application are freed but the outputs produced by the application are still in place. The Job service stores the configuration of this completed application. It

additionally has the links to the outputs produced and the inputs used by the application and are available to the user when he/she desires to view them. The job service is another database oriented service (Section 4.5.1). It provides *newJob*, *getJobListByUser* and *getJobInfo* operations to add a new job entry, get all user jobs and to get the application configuration used for a particular job.

4.3 Marketplace services

Based on the expected interaction between the application providers, the users and the application market (Figure 6), this work identifies a number of services essential for its success. The major services include:

1. Metadata service: This service serves as the entry point to the application lifecycle. The application providers interact with the metadata service to register their applications and the application users access this service to browse and select applications. The abstract state application object is stored in this service.
2. Workspace services: Ready applications (configured from abstract applications) are preserved by this service. The service provides personal space for each user to store their configurations. The applications are now ready to be submitted.
3. Submission and file transfer services: The submission service is transient and receives a ready application with runtime parameters. The application now transitions to its active state. The application is submitted and managed on behalf of the user. The file transfer service is used by the submission service as needed.

4.Persistence service: The ghost application is preserved by the persistence service.

This application state is created by the submission service so as to preserve the application run long after submission is complete and the submission service has been removed from the system.

Of these services, metadata, workspace and persistence are database oriented services – the services primarily concerned with database store, search and retrieve operation. The submission service, the most important service of all, directly interacts with the backend machine.

Of these services, the metadata service is accessed by both the application providers and the users (Figure 9). It stores captured applications from the application providers and makes these captured applications available to the users. The persistence services store information about all the user job runs along with their run configuration thus saving the user the burden of manually maintaining a job journal. The workspace services provide user space to store a user's personal job configuration in a convenient hierarchical directory-like structure. The last service, the submission service is responsible for a user's job submissions. It interacts with the backend machines on behalf of the user.

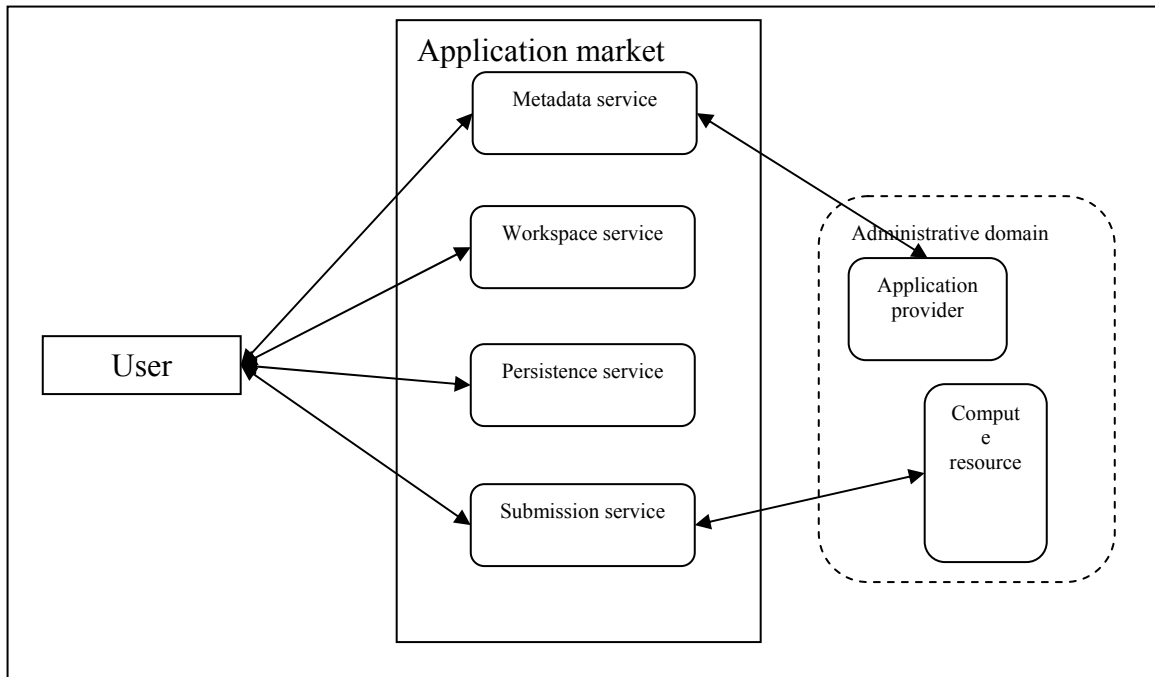


Figure 9: CPU Market middleware

4.4 The Multi-transport architecture

4.4.1 Service implementation

The application marketplace requirements mandate that services be offered using many RPC based mechanisms. For this reason, the application market reuses the multi protocol architecture that was developed for the DMEFS project. Such an implementation ensures that the same services are accessible using multiple protocols. It also ensures that newer and better RPC mechanisms formulated in future could be readily incorporated into the marketplace.

The multi-protocol architecture permits the same service to be accessed using different protocols (

Figure 10). Best effort is made to ensure that the same source base is used and different packaging tools are applied to package the services to their respective hosting environments. For example, a client who prefers to access the application market using OGSA services contacts and interacts with the OGSA server using GWSDL, SOAP 1.1 and HTTPG. In this case, just the service stubs generated from the service interfaces is hosted in the OGSA container. These stubs then forward service requests to the business logic implementation. The service business logic and database connectivity beans are hosted as EJBs on a separate container and these EJBs interact with the database.

Implementing the application market services as OGSA services has many advantages.

The advantages include:

- 1.Ability to authenticate the user using his secure grid certificate and provide a secure channel at the same time.
- 2.Ability to create web service instances which is a requirement for an extendable submission service.

Needless to say, providing the application market services as OGSA services is the preferred implementation and the major contribution of this work. The rest of this chapter, with the exception of the digression regarding multi protocol support on the client side, is primarily concerned with OGSA services.

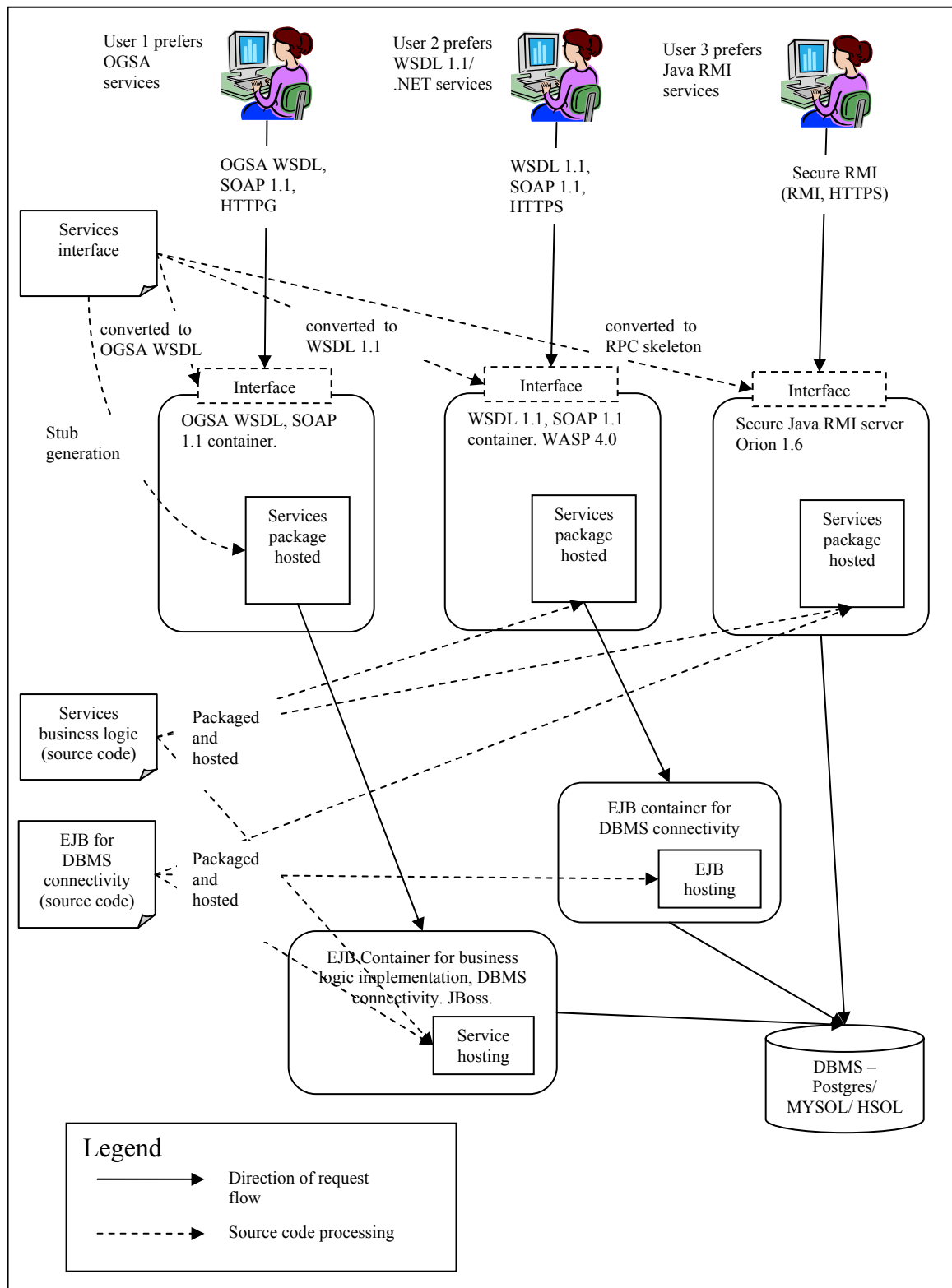


Figure 10: Multi-protocol architecture for database oriented services

4.4.2 Client implementation

The client implementation is at liberty to use any of the protocols supported by the application market. While it is not a requirement for a single client to support multiple protocols, this work chose to use a client that could use different protocols to access the application market. The user selects the actual protocol that would be used by the client to interact with the application market.

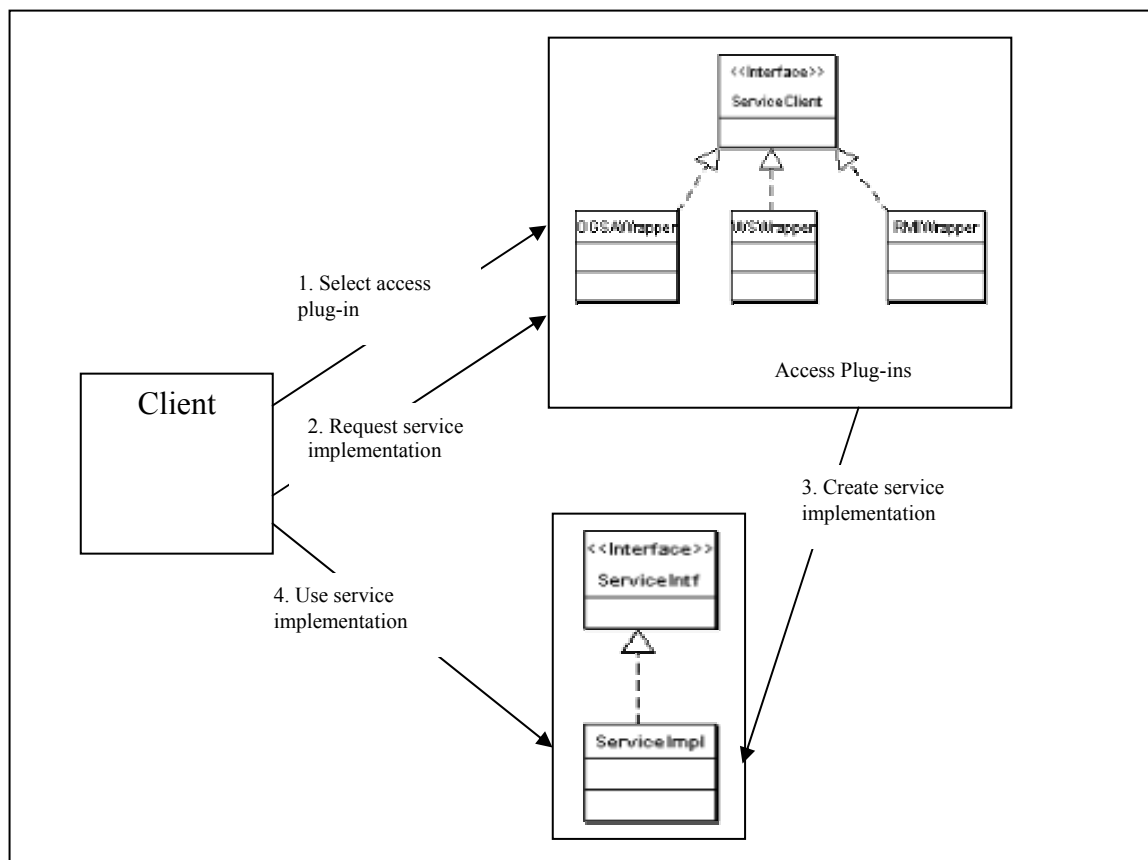


Figure 11: Client multi-protocol implementation

The multi-protocol client is implemented using a “plug-in” access service design. In this design, each protocol provides its own “plug-in” which implements the *ServiceClient*

interface. The *ServiceClient* defines the *getServiceHandle* operation that takes the name of the service as an argument and returns a reference to the stub that implements the protocol and has the same interface as the service. This is a modified implementation of the Proxy software design pattern [51]. Since different protocol stubs for a service implementation provide the same service interface, the client is hidden from the actual protocol differences. Once a service stub is bound to the service interface, the client invokes operations on the service using its well defined interface oblivious of the actual mechanisms used to implement the operations. At present, this architecture supports three different transport mechanisms

1. GWSDL/HTTPG implementation using Globus toolkit 3.0
2. WSDL 1.1/HTTPS implementation using Wasp toolkit 4.0.
3. Java RMI/HTTPS implementation using Orion 1.6 [52].

4.5 CPU Market services implementation

One of the goals of this work is to produce an implementation that is not tied to any specific container or service provider environment. Hence, as far as possible, services are written such that they conform to a specification such as the EJB or the OGSA and they can be used on any container that conforms to these specifications.

4.5.1 Database oriented services

Though database-oriented services and non-database oriented services are provided using the same protocols, differences between the services they provide require fundamentally different implementations. The differences stem from the fact that the database-oriented

services are essentially stateless, whereas the submission service, a non-database oriented service requires stateful services.

Database-oriented services have two components or tiers – the business logic tier and the database tier. The database tier has two subcomponents – the data objects and the database [2]. Such an implementation confirms to proper software design by using reusable components. Moreover, all three components can be hosted on physically different machines, thus making the services scalable, shifting the burden across multiple machines. In this design, the components could be implemented as standalone processes or as packages hosted by a container. This work chose to follow the second approach and implements the business tier and the data objects as packages hosted by a J2EE container (the database is still a separate process though). The business tier has stateless services; hence, this tier is implemented using stateless session EJBs. The session beans interact with the data objects, which are implemented using entity EJBs. The entity EJBs use container-managed persistence to operate on the database. Since the J2EE container that hosts the business logic mandates that the services be accessed using Java RMI, this design cannot be used “as such” to create web services. Fortunately, the Globus 3.0 toolkit provides tools to project services hosted in an EJB container as web services, which is the approach used in this work (Section 2.4.1).

4.5.2 The submission service

The submission service is the most important component of the application market. The functions it provides includes:

- Support for job submission
- Support for reservation
- Job management
- Job status notification
- Automatic user job history updating.

The marketplace uses a web service factory to implement the submission service. Each job submission is handled by a new submission service instance that is created “on demand” by the factory. Using a factory to create job service instances has multiple advantages:

- The user proxy or credentials can be cached by the service instance to act on behalf of the user.
- The user can directly contact the service instance for his control requests.
- Properties of the current job that is being executed can be stored as the state of the submission service instance.
- Each submission instance handles its own notifications.

Many implementations have been suggested to create such a factory and instances. For example, the instances could be implemented as new operating system processes or threads. This work uses the factory service implementation included with the Globus 3.0 toolkit. In this implementation, a new grid service instance request spawns a grid service with a uniquely locatable GSH. It which comes into existence when a job needs to be submitted by the user and it goes out of existence when the job is complete. During its

lifetime, the instance does all functions on behalf of the user. The submission instance additionally requires notification mechanisms to notify the user regarding job status changes.

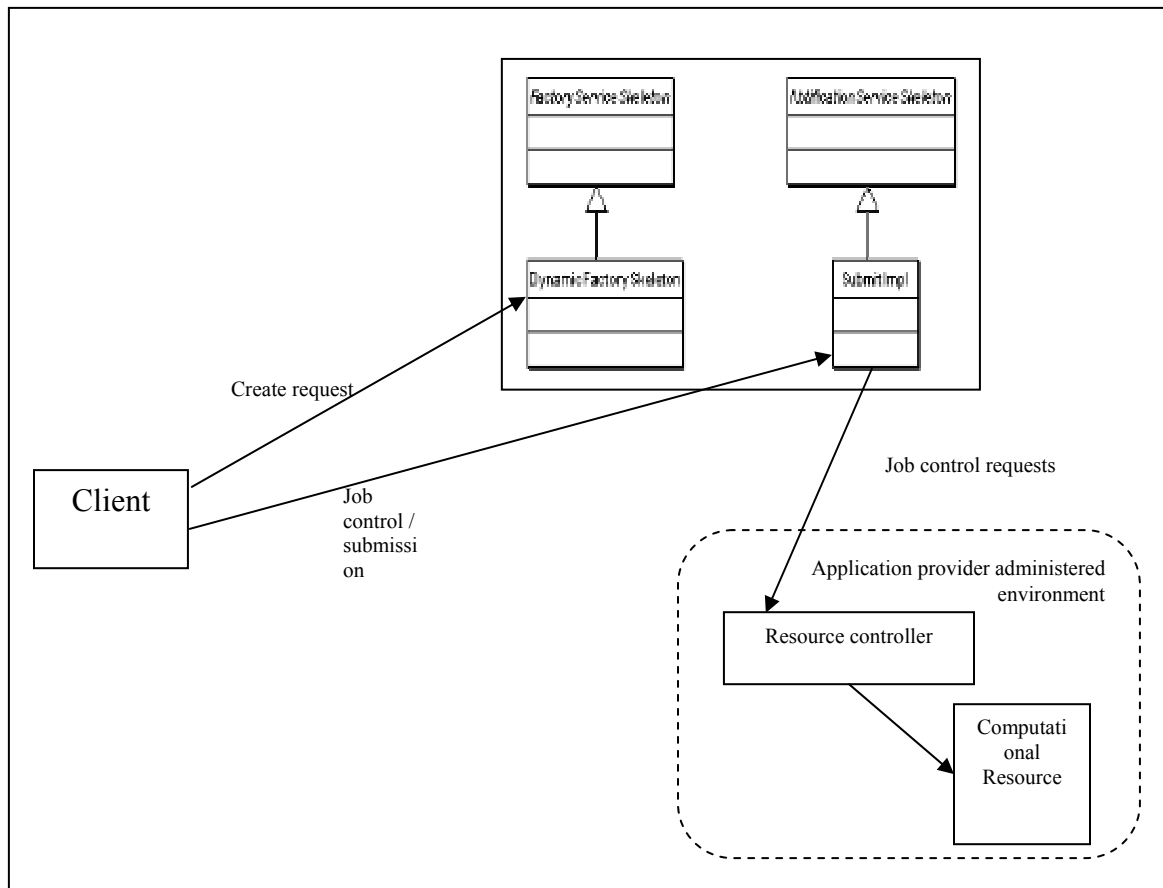


Figure 12: Submission service implementation

Thus, the current application market service implementation uses the *factory* and *notification* port types to implement the submission service. The factory is used to request a new submission service instance that implements all functions required for job submission and control. The submission service instance, in turn, implements the notification port type to notify listeners regarding job status changes (Figure 12).

Additionally, complete job information (including its status) is available as service data of the submission instance.

In a typical usage scenario, the client first requests that the submission factory create a new submission service for the user's request. Once a new submission service instance is created, the client sends the user request which includes the job configuration information, batch parameters as well as QOS requirements along with a user proxy (his credentials). The submission instance determines the type of job control mechanisms (currently globus 2.x and globus 3.x) to use depending on the job configuration and selects one for handling the current job. Once this selection is made, it makes reservations to satisfy user QOS requirements (if possible). Next, the job request is submitted by constructing the older Globus 2.x based RSL or the newer XML based RSL for Globus 3.x and invoking submission mechanisms as required.

CHAPTER V

RESULTS

5.1 Separation of concerns in the marketplace

The marketplace combines the requirements of two classes of individuals – the providers and the users, and creates an environment that is beneficial to both. The providers needed an infrastructure to publish their applications and computational resources, and the users needed a simple interface to browse, configure, run and maintain journals of applications. The requirements of these two classes of individuals are independent and the marketplace caters to the needs of both by providing a different interface to these classes of users (Figure 13).

The providers now have an interface to publish and modify applications that they want to “advertise” on the market place. They are responsible for keeping application information up-to-date and that all information necessary to configure and run the application are included as a part of the flexible metadata that describes the application. The user, on the other hand, is relieved from knowing the application intricacies, which is a concern of the provider.

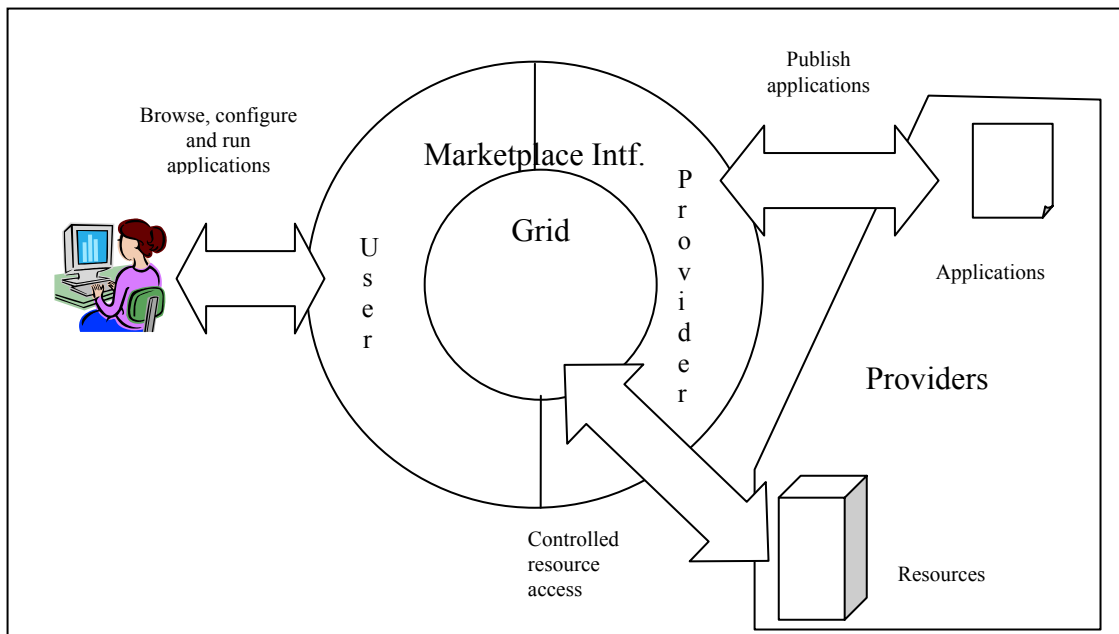


Figure 13: Separate interfaces for providers and users

The user (who interacts using the client), in turn, receives the application metadata that describes all applications in a single uniform format. It can be introspected to know all information necessary to configure and run the application. The uniform format of the metadata also permits a GUI to be built to automate configuration and submission tasks, thus, allowing the client to present an easy-to-use interface to the user. The steps the user follows to run his application are now reduced to the following:

1. Select an application from the list of applications
2. Use the simple interface to configure the application: provide application arguments, parameters, input/output file locations and parameters for batch submission. The interface remains simple irrespective of the complexity of the application. If the provider chooses to provide his own GUI, it can be accommodated too.

3. Save the configuration, submit it to the batch queue or run the application interactively.

5.2 Application provider's view

The application provider (along with the CPU provider) is responsible for the description of his applications in the marketplace. The description was required to contain all information necessary to locate, configure and run the application. It additionally had to be represented in a format that is friendly to all possible platforms and architectures the user could possibly choose. Hence, this work chooses the web friendly text based XML format to capture the application description. The schema (grammar, Appendix 1) of this XML format is designed to capture all necessary information about the application, which includes the components shown in Table 3.

Table 3: Application description components

- | |
|---|
| <ol style="list-style-type: none"> 1.A textual description of the application which includes references to application support (if necessary). 2.Machine specific and independent arguments. 3.Names and location of the input, parameter and output files 4.Environment variables required by the application. 5.Default QOS parameters for the application. 6.Names of the computational machines that host the application and corresponding application location information. |
|---|

Such an application description (An example shown in Appendix II) can be “viewed” by a user or interpreted by a client to know details about the application. The description was, in part (the QOS section of the description was enhanced for this work), used in the DMEFS project to describe and consequently run renowned “complex” applications like COAMPS [53] and NCOM [54]. The DMEFS project was demonstrated at the DMEFS workshop (March 2003).

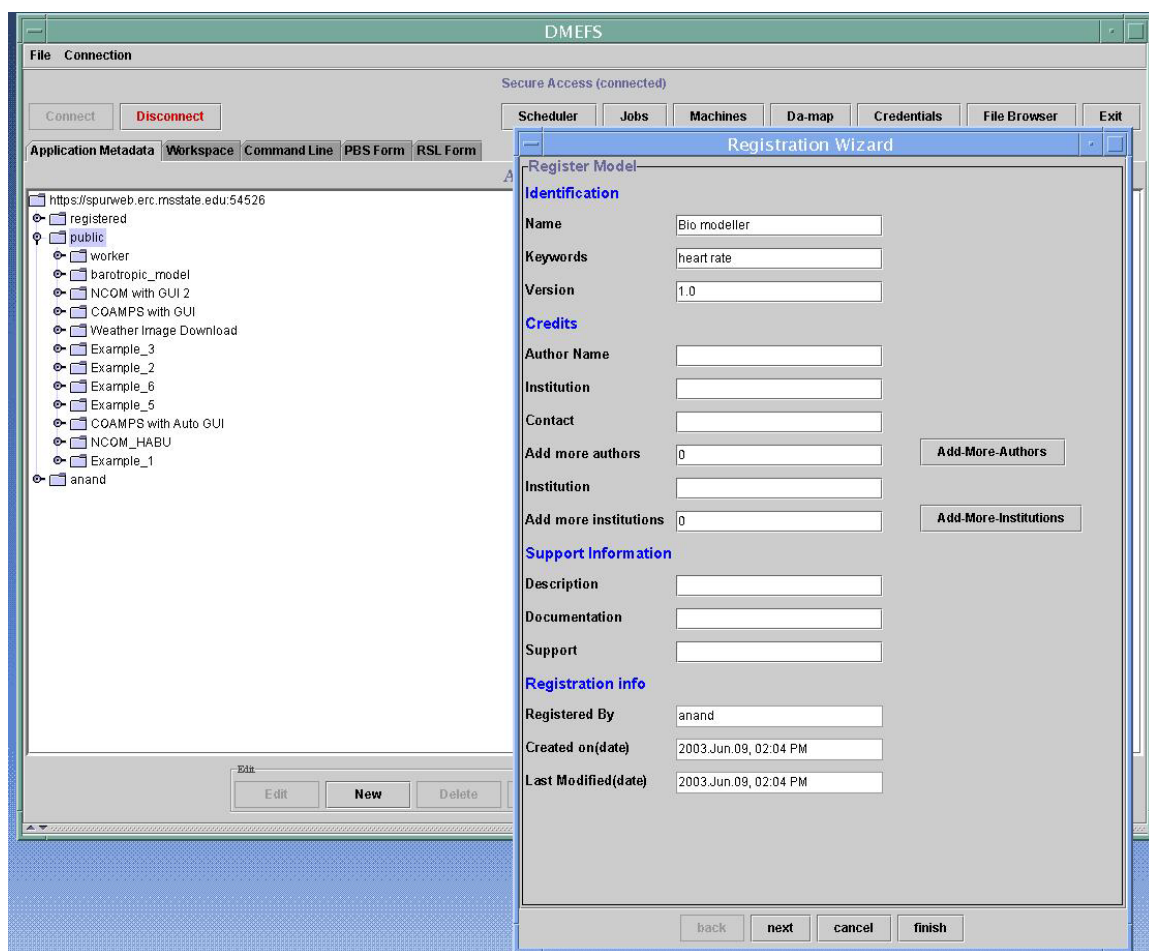


Figure 14: Application registration GUI

The XML application description, though human readable, is too tedious and error prone to be scripted by hand. Hence, this work provides a GUI (originally a part of DMEFS) to collect application information from the providers, convert it to XML and add it to the marketplace. The GUI (Figure 14) conveniently hides the *newModel* and *newHost* grid web service invocations that are used to add the application description to the marketplace. Thus, the provider, who is responsible for the description of his applications, uses the easy-to-use interface to add his applications and keep them updated at the marketplace.

5.3 User view

The user, on the other hand, can access the application description that contains all information necessary to configure and run the application. The user requirements included:

1. Mechanisms to conveniently browse and access application descriptions.
2. Uniform mechanisms (that hide application and computational resource intricacies) to configure and submit applications.
3. A “personalized” web service to manage his/her job.

Consequently, this work provides four services – metadata, workspace, persistence and submission. All services provide convenient GUIs (initially a part of the DMEFS project) to present an intuitive interface to the user. The metadata service (which also serves the application providers) supports *getAllModelsList*, *getModelInfo* and *getHostInfo* grid service functions to browse and retrieve application descriptions. The application object,

now in its abstract state, needs to be customized for the user requirements. Once an application is selected, the generality and XML format of the application description, along with XML tools (Section 2.5), is used to create a GUI to configure (or customize) the application.

The configuration step allows the user to customize all options (Table 3) that the providers choose to reveal about the application. The GUI conveniently hides XML processing from the user, while, at the same time, providing an interface with configuration options and descriptions originally “described” by the provider. Once configured, the application object, now in its ready state, could be optionally stored with the workspace service or be submitted.

Since the user requires constant control over his job with optional notification regarding job status changes, submission is handled by a grid service factory. The submission service which implements the factory port type spawns transient submission instances that are responsible for job control and notification. The submission instance analyzes the QOS requirements of the user and performs any necessary reservations to satisfy the same (at preset, CPU reservation is the only QOS supported). If the QOS requirements are satisfied, the job is submitted and the application object is now “active.” The submission instance web service provides functions for job control including submission and termination. It can notify the user regarding job status changes and will automatically be destroyed when the job is complete.

Before the submission instance exits, it updates the job information at the persistence service that stores a journal of the user's job runs. The pedigree of the application with configuration and links to the input/output files is available with the persistence service for reference at a later date.

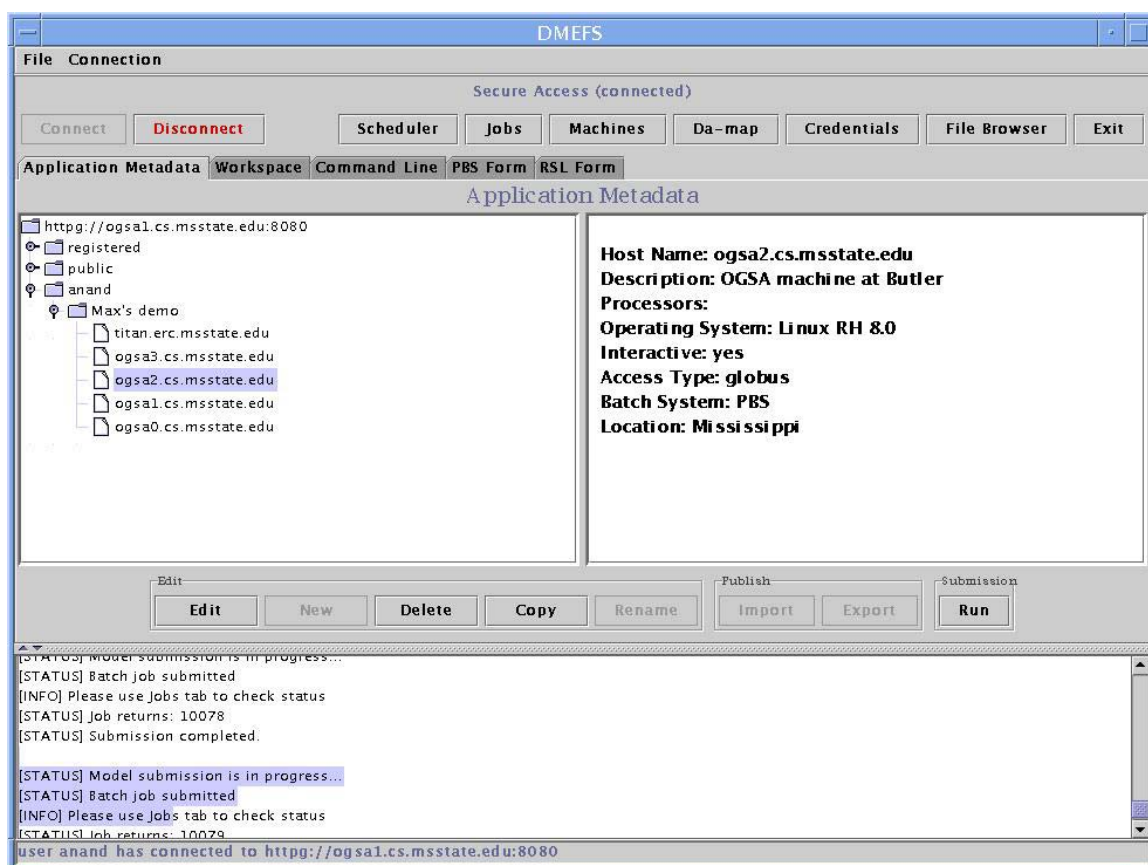


Figure 15: Application list

Configuration

Model dependent config | **Run configuration** | Batch parameters

Submission type

Queue: pbsdef

Run Parameters

Max time:

Min time:

Max wall time: 0:30:00

Max CPU time:

Max memory: 512M

Min memory:

Host count: 1

Processor count: 1

Gram job:

Dry run: N

Project group:

QOS parameters

Start time: 3:00:00

Cancel Back Run Next

Figure 16: Application configuration with CPU time QOS

5.4 Hypothesis validation

The application market place is an infrastructure that caters to the requirements of application providers, CPU providers and users. Its services, all based on OGSA (Globus toolkit 3.0 implementation), provide an Internet-friendly, secure, architecture independent mechanism to provide distinct interfaces to the providers and the users. The providers and the users play distinct roles in the marketplace – the providers advertise and update information about applications and the users use these application descriptions to select, configure and run applications.

The application providers and the CPU providers use the market place metadata service and the convenient GUI service access to “advertise” all qualities of the application pertinent to the user. The application description thus obtained from the user is transparently encoded to XML, the “universal format for data on the web” [55] for it to be stored in the metadata service and introspected on any client. The schema of the application description was verified to be capable of describing complex applications (Section 5.2). Consequently, the second hypothesis that it is possible to capture a computational application in a portable format can be claimed proven.

The marketplace provides a variety of services that revolve around the lifecycle of the application to satisfy the user requirements. The metadata service originates the application lifecycle by providing a list of applications and their descriptions. The uniform schema used to describe applications permits the user to configure the application using an intuitive GUI wizard. Once configured, the application can be submitted using a generic submission service factory that creates transient submission grid service instances to manage user jobs. The instance submission services are themselves driven by the configuration information and provide job control and notification after submission. Thus, the claim of the first hypothesis that the users can have a convenient market place to select, configure and run applications using the instance service based infrastructure with notification can be proclaimed proven.

CHAPTER VI

FUTURE WORK

The application marketplace is still a concept in its formulation stage. Aspects that require enhancements include

- Support for chargeable grid services
- Enabling Kerberos as an authentication mechanism

Current support for compensating the providers is based on the “project” specified as a part of batch submission. The CPU time requested is charged against the project. The marketplace does not take part in actual mechanisms used to translate the project CPU usage to remuneration. Using GESA (Section 2.7) enabled grid services could enable the marketplace to broker financial compensations for resource providers. It would also increase the user’s choice of applications.

The GSS API used by the submission service for authentication and encryption uses globus credentials for its current implementation. This limits the accessibility of submission service instances to globus enabled resources. GSS API inherently supports Kerberos [56] and enabling Kerberos authentication would increase the variety of resources accessible through the marketplace.

Though the concept of marketplace could support multiple services provided by different (distributed) portals, the current implementation demonstrates just a single set of centralized services. The current implementation could be extended to support distributed services with provision for finding and accessing applications across portals.

The current work is based on a pre release of Globus 3.0. This restricts usable security mechanisms to access to EJB services to transport level security (and not message level security). Upgrading implementation to the latest release should enable message level security for all services.

REFERENCES

- [1] I Foster and C. Kesselman, "Globus: A Metacomputing infrastructure toolkit," The International Journal of Supercomputer Applications and High Performance Computing, vol. 11, no. 2, summer 1997, pp. 115-128.
- [2] I Foster and C. Kesselman, "The Globus project: a status report," Proc. Heterogeneous Computing Workshop, 1998. (HWC 98), pp. 4 – 18.
- [3] J. Hart, "How web services came to be,"
<http://www.webservicesarchitect.com/content/articles/hart01.asp>
- [4] D. Box, D. Ehnebuske, G. Kalkivaya, A Layman, N Mendelsohn, H. F. Nielsen, S. Thatte, D. Winer, "Simple Object Access Protocol," W3C Note,
<http://www.w3.org/TR/SOAP/>.
- [5] E. Christensen, F. Curbera, G. Meredith and S. Weerawarana, "Web Services Description Kanguage (WSDL) 1.1," W3C Note, <http://www.w3.org/TR/wsdl>.
- [6] <http://www-106.ibm.com/developerworks/library/ws-start.html>
- [7] <http://www.globus.org/ogsa/>
- [8] <http://www.gridforum.org/>
- [9] P. J. Martin, "A description of the Navy Costal Ocean Model Version 1.0," tech. Memo, NRL.
- [10] P. J. Martin, "NCOM user guide for NCOM 1.3," user guide, NRL.
- [11] http://www.erc.msstate.edu/npebc/seminars-f02/haupt_npebc.pdf
- [12] <http://www.w3c.org/2002/ws/>
- [13] HTTP, <http://www.w3.org/Protocols/>
- [14] K Cagle, J Duckett, O Griffin, S Mohr, F Norton, N Ozu, I. Stokes-Rees and K. Williams, Professional XML Schemas, Brimingham, UK: Wrox Press Ltd. 2001
- [15] www.webservices.org

- [16] K. Apshankar, H. Chang, M. Clark, E. B. Fernandez, P. Fletcher, W. Hankison, J. J. Hanson, R. Irani, K. Mittal, J. M. Myerson, D. O’Riordan, D. Sadhwani, G. Samtani, B. Siddiqui, J. Thelin, M. Waterhouse, C. Wiggers, L. Zhang, “Web Services Business Strategies and Architectures,” Chicago: Expert Press. (web service architectures)
- [17] A. Kalyanasundaram and T. Haupt, “Using secure web services for development for a grid computing environment,” High Performance Computing Symposium, 2003.
- [18] <http://www.globus.org/gt2/GRAM.html>
- [19] <http://www.globus.org/datagrid/gridftp.html>
- [20] http://www.globus.org/about/events/US_tutorial/slides/Dev-08-Information1.pdf
- [21] <http://www.globus.org/security/>
- [22] <http://www.globus.org/datagrid/deliverables/C2WPdraft3.pdf>
- [23] http://www.globus.org/tutorial/slides/User%20html/user_03_mds/tsld019.htm
- [24] http://www.globus.org/tutorial/slides/User%20html/user_03_mds/tsld020.htm
- [25] <http://www-unix.globus.org/ogsa/docs/alpha/>
- [26] S. Tueke, K. Czajkowski, I. Foster, J. Frey, S. Graham, C. Kesselman, T. Maquire, T. Sandholm, D. Snelling and P. Vanderbilt, “Open Grid Services Infrastructure version 1.0.” tech. memo, Global Grid Forum, 2003, http://www-unix.globus.org/toolkit/draft-ggf-ogsi-gridservice-33_2003-06-27.pdf.
- [27] <http://www-106.ibm.com/developerworks/grid/library/gr-gt3/>
- [28] <http://jakarta.apache.org/tomcat/>
- [29] E. A. Jezierski, G. Malcolm and L. Joyner, Application architecture for .NET: Designing Applications and Services. Microsoft 2001.
- [30] <http://www.ibm.com/websphere>
- [31] <http://www.jboss.org/>
- [32] L. Marco, EJB & JSP: Java on the edge. John Wiley & Sons October, 2001.

- [33] S. H. Simon, "XML," New York: Mc Graw-Hill, 2001.
- [34] B. Mc Laughlin, "Java and XML," Sebastopol, CA: O'Reilly & Associates, Inc 2000.
- [35] <http://www.jdom.org/>
- [36] <http://cairo.cs.uiuc.edu/software/DSRT-2/dsrt-2.html>
- [37] P. Goyal, H. M. Vin and H. Cheng, "Start-time Fair Queueing: A scheduling Algorithm for Integrated Services Packet Switching Networks," <http://www.cs.utexas.edu/users/vin/pub/pdf/sigcomm96.pdf>
- [38] R. Braden, L. Zhang, S. Berson, S. Herzog and S. Jamin, "Resource ReSerVation Protocol (RSVP)," Network Working Group RFC 2205, Sept 1997, <ftp://ftp.isi.edu/in-notes/rfc2205.txt>.
- [39] <http://www.cs.utexas.edu/users/vin/pub/pdf/CelloExtended.pdf>
- [40] A. Roy, "End-to-end Quality of Service for High-End Applications," doctoral dissertation, Dept. Computer Sciences, University of Chicago 2001.
- [41] <http://www.ggf.org/Meetings/ggf7/drafts/CompEconArch1.pdf>
- [42] <http://www.doc.ic.ac.uk/~sjn5/GGF/gesa-wg.html>
- [43] <http://edocs.bea.com/workshop/docs81/doc/en/core/index.html>
- [44] M. Thomas, S. Mock, M. Dahan, K. Mueller, D. Sutton, J. R. Boisseau, "The GridPort toolkit: a system for building Grid portals," Proc. 10th IEEE International Symposium, 2001, pp 216 – 227.
- [45] P. Bangalore, "An Open Framework for Developing Distributed Computing Environments for Multidisciplinary Computational Simulations." doctoral dissertation, Dept. Computational Engineering, Mississippi State University 2003.
- [46] http://www.systinet.com/products/java_ws
- [47] I Foster, C. Kesselman and S Tuecke, "The Anatomy of the Grid: Enabling Scalable Virtual Organizations," Proc Cluster Computing and the Grid, 2001, pp 6-7.
- [48] R. McCormack, J. Koontz and J. Devaney, "Seemless Computing with WebSubmit," Concurrency : Practice and Experience 1999, vol. 11, no. 15.

- [49] <http://www.globus.org/>
- [50] T. Haupt, "Grid Job and Distributed Simulation Systems," Information paper submitted to the Job Submission Description Language GGF Working Group.
- [51] Applying UML and Patterns, Craig Larman. Prentice Hall 1998.
- [52] <http://www.orionserver.com/>
- [53] http://www.fnoc.navy.mil/PUBLIC/MODEL_REPORTS/MODEL_SPEC/coamps2.0.html
- [54] http://www7320.nrlssc.navy.mil/global_ncom/
- [55] <http://msdn.microsoft.com/library/default.asp?url=/nhp/default.asp?contentid=28000438>
- [56] <http://java.sun.com/j2se/1.4.1/docs/guide/security/jgss/single-signon.html>

APPENDIX A
APPLICATION DESCRIPTOR XML SCHEMA

```

<?xml version="1.0" encoding="UTF-8"?>
<!-- edited with XMLSPY v2004 rel. 2 U (http://www.xmlspy.com) by Anand (K) --
>
<!--W3C Schema generated by XMLSPY v2004 rel. 2 U
(http://www.xmlspy.com)-->
<xs:schema xmlns:xs="http://www.w3.org/2001/XMLSchema"
elementFormDefault="qualified">
  <xs:element name="AD">
    <xs:annotation>
      <xs:documentation>Application Descriptor root
Element</xs:documentation>
    </xs:annotation>
    <xs:complexType>
      <xs:sequence>
        <xs:element ref="application"/>
        <xs:element ref="target"/>
      </xs:sequence>
    </xs:complexType>
  </xs:element>
  <xs:element name="QOS">
    <xs:complexType>
      <xs:sequence>
        <xs:element ref="cpu"/>
        <xs:element ref="memory"/>
        <xs:element ref="adaptionrule"/>
        <xs:element name="environment" type="envdef"/>
      </xs:sequence>
    </xs:complexType>
  </xs:element>
  <xs:element name="adaptionrule" type="xs:anyType"/>
  <xs:element name="application">
    <xs:annotation>
      <xs:documentation>Machine independent information regarding
application</xs:documentation>
    </xs:annotation>
    <xs:complexType>
      <xs:sequence>
        <xs:element ref="signature"/>
        <xs:element ref="description"/>
        <xs:element ref="documentation"/>
        <xs:element ref="support"/>
        <xs:element ref="reginfo"/>
        <xs:element ref="arguments"/>
        <xs:element ref="parameterfiles"/>
        <xs:element ref="inputfiles"/>
        <xs:element ref="outputfiles"/>

```

```

        <xs:element ref="gui"/>
        <xs:element ref="QOS"/>
        <xs:element ref="source"/>
    </xs:sequence>
    <xs:attribute name="id" type="xs:ID" use="required"/>
</xs:complexType>
</xs:element>
<xs:element name="argument">
    <xs:complexType>
        <xs:sequence>
            <xs:element ref="name"/>
            <xs:element ref="description"/>
            <xs:element ref="type"/>
            <xs:element ref="restrictions"/>
            <xs:element ref="value"/>
            <xs:element ref="order"/>
        </xs:sequence>
        <xs:attribute name="id" type="xs:ID" use="required"/>
        <xs:attribute name="multiplicity" type="xs:boolean" use="required"/>
        <xs:attribute name="syntax" type="xs:string" use="required"/>
    </xs:complexType>
</xs:element>
<xs:element name="arguments">
    <xs:complexType>
        <xs:sequence>
            <xs:element ref="argument" minOccurs="0"
maxOccurs="unbounded"/>
        </xs:sequence>
    </xs:complexType>
</xs:element>
<xs:element name="author">
    <xs:complexType>
        <xs:sequence>
            <xs:element ref="name"/>
            <xs:element ref="institution"/>
            <xs:element ref="contact"/>
        </xs:sequence>
    </xs:complexType>
</xs:element>
<xs:element name="build" type="xs:string"/>
<xs:element name="class" type="xs:string"/>
<xs:element name="contact" type="xs:string"/>
<xs:element name="count" type="xs:integer"/>
<xs:element name="cpu">
    <xs:complexType>
        <xs:sequence>

```

```

        <xs:element name="min" type="xs:duration"/>
        <xs:element name="max" type="xs:duration"/>
        <xs:element name="endtime" type="xs:dateTime"/>
    </xs:sequence>
</xs:complexType>
</xs:element>
<xs:element name="created" type="xs:dateTime"/>
<xs:element name="credit">
    <xs:annotation>
        <xs:documentation>Application provider's
information</xs:documentation>
    </xs:annotation>
    <xs:complexType>
        <xs:sequence>
            <xs:element ref="author" minOccurs="0" maxOccurs="unbounded"/>
            <xs:element ref="institution" minOccurs="0"
maxOccurs="unbounded"/>
        </xs:sequence>
    </xs:complexType>
</xs:element>
<xs:element name="cvsroot" type="xs:token"/>
<xs:element name="description" type="xs:string"/>
<xs:element name="order" type="xs:string"/>
<xs:element name="destmachine" type="xs:string"/>
<xs:element name="destname" type="xs:string"/>
<xs:element name="destpath" type="xs:string"/>
<xs:element name="documentation" type="xs:anyURI"/>
<xs:element name="dryrun">
    <xs:simpleType>
        <xs:restriction base="xs:token">
            <xs:enumeration value="yes"/>
            <xs:enumeration value="no"/>
        </xs:restriction>
    </xs:simpleType>
</xs:element>
<xs:complexType name="envdef">
    <xs:sequence>
        <xs:element ref="variable" minOccurs="0" maxOccurs="unbounded"/>
    </xs:sequence>
</xs:complexType>
<xs:complexType name="envval">
    <xs:sequence>
        <xs:element ref="value"/>
    </xs:sequence>
    <xs:attribute name="idref" type="xs:IDREF" use="required"/>
</xs:complexType>

```

```

<xs:element name="executable" type="xs:string"/>
<xs:complexType name="fileparam">
  <xs:sequence>
    <xs:element ref="parameter" minOccurs="0" maxOccurs="unbounded"/>
  </xs:sequence>
  <xs:attribute name="idref" type="xs:IDREF" use="required"/>
</xs:complexType>
<xs:complexType name="fileoutput">
  <xs:sequence>
    <xs:element ref="srcpath" minOccurs="0"/>
    <xs:element ref="srcname" minOccurs="0"/>
    <xs:element ref="destpath" minOccurs="0"/>
    <xs:element ref="destname" minOccurs="0"/>
    <xs:element ref="destmachine" minOccurs="0"/>
  </xs:sequence>
  <xs:attribute name="idref" type="xs:IDREF" use="required"/>
</xs:complexType>
<xs:complexType name="fileinput">
  <xs:sequence>
    <xs:element ref="srcpath" minOccurs="0"/>
    <xs:element ref="srcmachine" minOccurs="0"/>
    <xs:element ref="srcname" minOccurs="0"/>
    <xs:element ref="destpath" minOccurs="0"/>
    <xs:element ref="destname" minOccurs="0"/>
  </xs:sequence>
  <xs:attribute name="idref" type="xs:IDREF" use="required"/>
</xs:complexType>
<xs:complexType name="filedef">
  <xs:sequence>
    <xs:element ref="name"/>
    <xs:element ref="metadata"/>
    <xs:element ref="description"/>
  </xs:sequence>
  <xs:attribute name="id" type="xs:ID" use="required"/>
</xs:complexType>
<xs:element name="grammyjob">
  <xs:simpleType>
    <xs:restriction base="xs:token">
      <xs:enumeration value="collective"/>
      <xs:enumeration value="independent"/>
    </xs:restriction>
  </xs:simpleType>
</xs:element>
<xs:element name="gui">
  <xs:complexType>
    <xs:sequence>

```

```

        <xs:element ref="jsp"/>
        <xs:element ref="class"/>
        <xs:element ref="url"/>
    </xs:sequence>
</xs:complexType>
</xs:element>
<xs:element name="host" type="xs:token"/>
<xs:element name="hostcount" type="xs:integer"/>
<xs:element name="inputfiles">
    <xs:complexType>
        <xs:sequence>
            <xs:element name="file" type="filedef" minOccurs="0"
maxOccurs="unbounded"/>
        </xs:sequence>
    </xs:complexType>
</xs:element>
<xs:element name="inputs">
    <xs:complexType>
        <xs:sequence>
            <xs:element name="file" type="fileinput" minOccurs="0"
maxOccurs="unbounded"/>
        </xs:sequence>
    </xs:complexType>
</xs:element>
<xs:element name="institution" type="xs:string"/>
<xs:element name="jsp" type="xs:anyURI"/>
<xs:element name="keywords" type="xs:NMTOKENS"/>
<xs:element name="label" type="xs:string"/>
<xs:element name="lastModified" type="xs:dateTime"/>
<xs:element name="max" type="xs:time"/>
<xs:element name="maxcputime" type="xs:integer"/>
<xs:element name="maxmemory" type="xs:integer"/>
<xs:element name="maxtime" type="xs:integer"/>
<xs:element name="maxwalltime" type="xs:integer"/>
<xs:element name="memory">
    <xs:annotation>
        <xs:documentation>Specified in MB</xs:documentation>
    </xs:annotation>
    <xs:complexType>
        <xs:sequence>
            <xs:element name="min" type="xs:long"/>
            <xs:element name="max" type="xs:long"/>
        </xs:sequence>
    </xs:complexType>
</xs:element>
<xs:element name="metadata" type="xs:anyType"/>

```

```

<xs:element name="min" type="xs:time"/>
<xs:element name="minmemory" type="xs:integer"/>
<xs:element name="mintime" type="xs:integer"/>
<xs:element name="name" type="xs:string"/>
<xs:element name="outputfiles">
  <xs:complexType>
    <xs:sequence>
      <xs:element name="file" type="filedef" minOccurs="0"
maxOccurs="unbounded"/>
    </xs:sequence>
  </xs:complexType>
</xs:element>
<xs:element name="outputs">
  <xs:complexType>
    <xs:sequence>
      <xs:element name="file" type="fileoutput" minOccurs="0"
maxOccurs="unbounded"/>
    </xs:sequence>
  </xs:complexType>
</xs:element>
<xs:element name="parameter">
  <xs:complexType>
    <xs:sequence>
      <xs:element ref="name"/>
      <xs:element ref="description"/>
      <xs:element ref="label"/>
      <xs:element ref="type"/>
      <xs:element ref="restrictions"/>
      <xs:element ref="value"/>
      <xs:element ref="order"/>
    </xs:sequence>
    <xs:attribute name="id" type="xs:ID" use="required"/>
  </xs:complexType>
</xs:element>
<xs:element name="parameterfiles">
  <xs:complexType>
    <xs:sequence>
      <xs:element name="file" type="fileparam" minOccurs="0"
maxOccurs="unbounded"/>
    </xs:sequence>
  </xs:complexType>
</xs:element>
<xs:element name="path" type="xs:token"/>
<xs:element name="project" type="xs:string"/>
<xs:element name="queue" type="xs:token"/>
<xs:element name="reginfo">

```



```

<xs:complexType>
  <xs:sequence>
    <xs:element ref="registeredBy"/>
    <xs:element ref="created"/>
    <xs:element ref="lastModified"/>
  </xs:sequence>
</xs:complexType>
</xs:element>
<xs:element name="registeredBy" type="xs:string"/>
<xs:element name="restrictions" type="xs:anyType"/>
<xs:element name="run">
  <xs:annotation>
    <xs:documentation>Machine specific runtime
information</xs:documentation>
  </xs:annotation>
  <xs:complexType>
    <xs:sequence>
      <xs:element ref="argument" minOccurs="0"
maxOccurs="unbounded"/>
      <xs:element name="environment" type="envval" minOccurs="0"
maxOccurs="unbounded"/>
      <xs:element ref="inputs"/>
      <xs:element ref="outputs"/>
      <xs:element ref="executable"/>
      <xs:element ref="workdir"/>
      <xs:element ref="maxtime"/>
      <xs:element ref="mintime"/>
      <xs:element ref="maxwalltime"/>
      <xs:element ref="maxcputime"/>
      <xs:element ref="maxmemory"/>
      <xs:element ref="minmemory"/>
      <xs:element ref="queue"/>
      <xs:element ref="hostcount"/>
      <xs:element ref="count"/>
      <xs:element ref="grammyjob"/>
      <xs:element ref="dryrun"/>
      <xs:element ref="project"/>
      <xs:element ref="stdin"/>
      <xs:element ref="stdout"/>
      <xs:element ref="stderr"/>
    </xs:sequence>
  </xs:complexType>
</xs:element>
<xs:element name="signature">
  <xs:complexType>
    <xs:sequence>

```

```

        <xs:element ref="name"/>
        <xs:element ref="keywords"/>
        <xs:element ref="version"/>
        <xs:element ref="credit"/>
    </xs:sequence>
</xs:complexType>
</xs:element>
<xs:element name="source">
    <xs:complexType>
        <xs:sequence>
            <xs:element ref="host"/>
            <xs:element ref="cvsroot"/>
            <xs:element ref="path"/>
        </xs:sequence>
    </xs:complexType>
</xs:element>
<xs:element name="srcmachine" type="xs:string"/>
<xs:element name="srcname" type="xs:string"/>
<xs:element name="srcpath" type="xs:string"/>
<xs:element name="stderr" type="xs:anyURI"/>
<xs:element name="stdin" type="xs:anyURI"/>
<xs:element name="stdout" type="xs:anyURI"/>
<xs:element name="support" type="xs:string">
    <xs:annotation>
        <xs:documentation>Application support information</xs:documentation>
    </xs:annotation>
</xs:element>
<xs:element name="target">
    <xs:annotation>
        <xs:documentation>Machine dependent information regarding
application</xs:documentation>
    </xs:annotation>
    <xs:complexType>
        <xs:sequence>
            <xs:element ref="build"/>
            <xs:element ref="run"/>
        </xs:sequence>
        <xs:attribute name="id" type="xs:ID" use="required"/>
    </xs:complexType>
</xs:element>
<xs:element name="type" type="xs:QName"/>
<xs:element name="url" type="xs:anyURI"/>
<xs:element name="value" type="xs:string"/>
<xs:element name="variable">
    <xs:complexType>
        <xs:sequence>

```

```
        <xs:element ref="name"/>
    </xs:sequence>
    <xs:attribute name="id" type="xs:ID" use="required"/>
</xs:complexType>
</xs:element>
<xs:element name="version" type="xs:string"/>
<xs:element name="workdir" type="xs:string"/>
</xs:schema>
```

APPENDIX B

EXAMPLE APPLICATION DESCRIPTOR

```

<?xml version="1.0" encoding="UTF-8"?>
<AAD>
  <application id="0">
    <signature>
      <name>Bio Modeller</name>
      <keywords>Heart rate</keywords>
      <version>1.0</version>
      <credit>
        <author>
          <name>John Doe</name>
          <institution>MSU</institution>
          <contact></contact>
        </author>
        <institution></institution>
      </credit>
    </signature>
    <description/>
    <documentation/>
    <support/>
    <reginfo>
      <registeredBy>anand</registeredBy>
      <created>2002.Oct.30, 12:16 PM</created>
      <lastModified>2002.Oct.30,
PM</lastModified>
    </reginfo>
    <arguments>
      <argument>modelarg1</argument>
      <name>arg2</name>
      <description/>
      <type/>
      <restrictions/>
      <value>def2</value>
    </argument>
  </arguments>
  <parameterfiles>
    <file idref="infile1">
      <parameter id="param-1-1">
        <name>param1</name>
        <description>desc1</description>
        <label>label1</label>
        <type/>
        <restrictions/>
        <value>def1</value></parameter>
      <parameter id="param-1-2">
        <name>param2</name>
        <description>desc2</description>
        <label/>
        <type/>
        <restrictions/>
        <value>def2</value>
      </parameter>
    </file>
  </parameterfiles>

```

```

        </file>
</parameterfiles>
<inputfiles>
    <file id="infile1">
        <name>ifile1</name>
        <metadata>mdata-ifile1</metadata>
        <description>desc1</description>
    </file>
    <file id="infile2">
        <name>ifile2</name>
        <metadata>mdata-ifile2</metadata>
        <description>desc2</description>
    </file>
</inputfiles>
<outputfiles>
    <file id="outfile1">
        <name>ofile1</name>
        <metadata>mdata-ofile1</metadata>
        <description>desc1</description>
    </file>
    <file id="outfile2">
        <name>ofile2</name>
        <metadata>mdata-ofile2</metadata>
        <description>desc2</description>
    </file>
</outputfiles>
<gui>
    <jsp/>
    <class/>
    <url/>
</gui>
<QOS>
    <cpu>
        <min/>
        <max/>
        <endtime/>
    </cpu>
    <memory>
        <min/>
        <max/>
    </memory>
    <adaptionrule/>
    <environment>
        <variable id="modelenviron2">
            <name>DATA_HOME</name>
        </variable>
        <variable id="modelenviron1">
            <name>JAVA_HOME</name>
        </variable>
    </environment>
</QOS>
<source>

```

```

        <host/>
        <cvsrc/>
        <path/>
    </source>
</application>
<target id="titan.erc.msstate.edu">
    <build>build</build>
    <run>
        <argument id="hostarg1" multiplicity="1"
syntax=" ">
            <name>mach-arg1</name>
            <description/>
            <type/>
            <restrictions/>
            <value>def1</value>
        </argument>
        <environment idref="modelenviron2">
            <value>/var/data</value>
        </environment>
        <environment idref="modelenviron1">
            <value>/opt/java/jdk</value>
        </environment>
        <inputs>
            <file idref="infile1">
                <srcpath>/vulcan/var/data</srcpath>
                <srcmachine>vulcan.erc</srcmachine>
                <srcname>ifile-23.1</srcname>
                <destpath>/var/data</destpath>
                <destname>ifile-23.2</destname>
            </file>
            <file idref="infile2">
                <srcpath>/vulcan/var/data</srcpath>
                <srcmachine>vulcan.erc</srcmachine>
                <srcname>ifile2-23.1</srcname>
                <destpath>/var/data</destpath>
                <destname>ifile2-23.2</destname>
            </file>
        </inputs>
        <outputs>
            <file idref="outfile2">
                <srcpath>/var/data</srcpath>
                <srcname>ofile2-34.4</srcname>
                <destpath>/vulcan/var/data2</destpath>
                <destname>ofile2-34.5</destname>
                <destmachine>vulcan</destmachine>
            </file>
            <file idref="outfile1">
                <srcpath>/var/data</srcpath>
                <srcname>ofile-34.4</srcname>
                <destpath>/vulcan/var/data</destpath>
                <destname>ofile-34.5</destname>
                <destmachine>vulcan</destmachine>
            </file>
        </outputs>
    </run>
</target>

```

```

        </file>
    </outputs>
    <executable>exec</executable>
    <workdir>wd</workdir>
    <maxtime>36h</maxtime>
    <mintime>12h</mintime>
    <maxwalltime>48h</maxwalltime>
    <maxcputime>8h</maxcputime>
    <maxmemory/>
    <minmemory/>
    <queue>titan-ql</queue>
    <hostcount>4</hostcount>
    <count>4</count>
    <grammyjob/>
    <dryrun>no</dryrun>
    <project/>
    <stdin>/data/input</stdin>
    <stdout>/data/output</stdout>
    <stderr>/data/error</stderr>
</run>
</target>
</AAD>

```