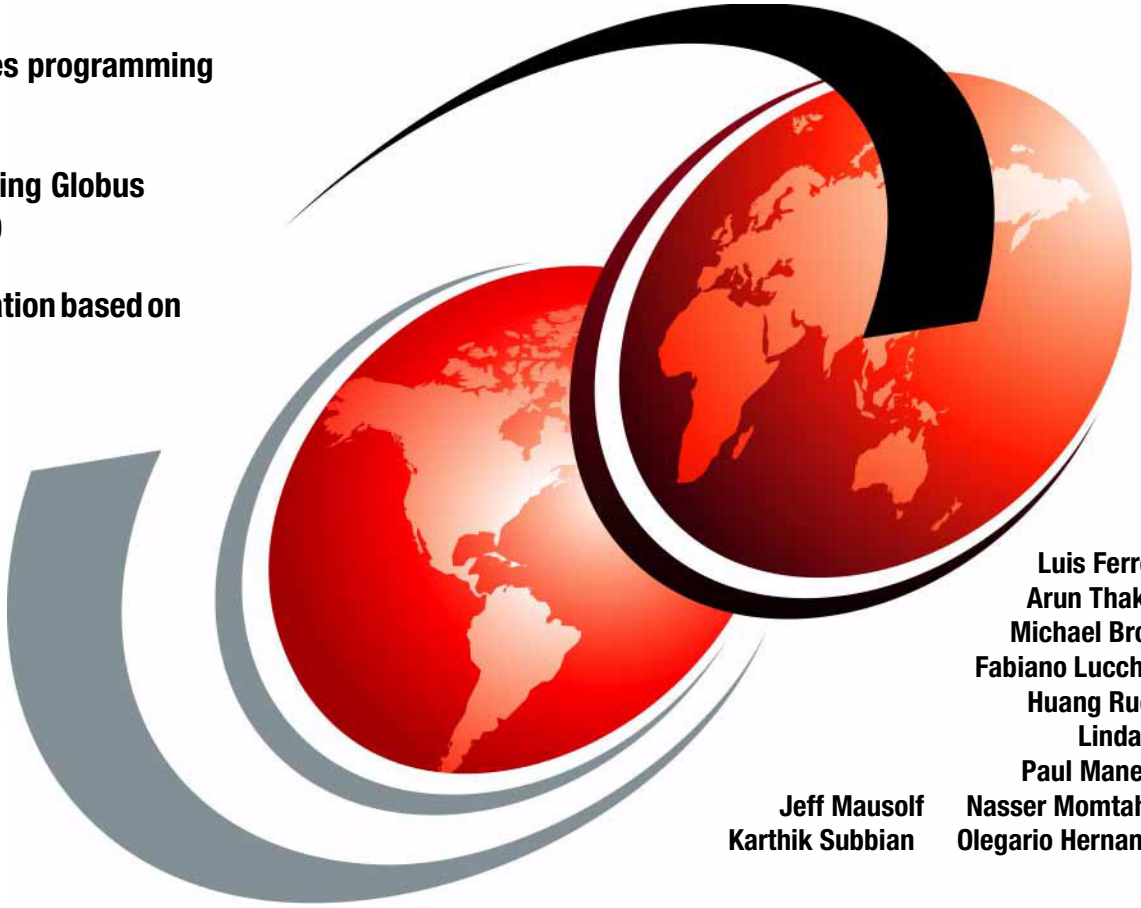
Grid Services Programming and Application Enablement

- Grid services programming
- Samples using Globus Toolkit V3.0
- Implementation based on OGSI V1.0



Luis Ferreira
Arun Thakore
Michael Brown
Fabiano Lucchese
Huang RuoBo
Linda Lin
Paul Manesco
Jeff Mausolf
Nasser Momtaheni
Karthik Subbian
Olegario Hernandez

ibm.com/redbooks

Redbooks

International Technical Support Organization

Grid Services Programming and Application Enablement

May 2004

SG24-6100-00

Note: Before using this information and the product it supports, read the information in “Notices” on page ix.

First Edition (May 2004)

This edition applies to version 3.0.2 of the Globus Toolkit and the OGSI 1.0 - Open Grid Services Infrastructure.

Contents

Notices

This information was developed for products and services offered in the U.S.A.

IBM may not offer the products, services, or features discussed in this document in other countries. Consult your local IBM representative for information on the products and services currently available in your area. Any reference to an IBM product, program, or service is not intended to state or imply that only that IBM product, program, or service may be used. Any functionally equivalent product, program, or service that does not infringe any IBM intellectual property right may be used instead. However, it is the user's responsibility to evaluate and verify the operation of any non-IBM product, program, or service.

IBM may have patents or pending patent applications covering subject matter described in this document. The furnishing of this document does not give you any license to these patents. You can send license inquiries, in writing, to:
IBM Director of Licensing, IBM Corporation, North Castle Drive Armonk, NY 10504-1785 U.S.A.

The following paragraph does not apply to the United Kingdom or any other country where such provisions are inconsistent with local law: INTERNATIONAL BUSINESS MACHINES CORPORATION PROVIDES THIS PUBLICATION "AS IS" WITHOUT WARRANTY OF ANY KIND, EITHER EXPRESS OR IMPLIED, INCLUDING, BUT NOT LIMITED TO, THE IMPLIED WARRANTIES OF NON-INFRINGEMENT, MERCHANTABILITY OR FITNESS FOR A PARTICULAR PURPOSE. Some states do not allow disclaimer of express or implied warranties in certain transactions, therefore, this statement may not apply to you.

This information could include technical inaccuracies or typographical errors. Changes are periodically made to the information herein; these changes will be incorporated in new editions of the publication. IBM may make improvements and/or changes in the product(s) and/or the program(s) described in this publication at any time without notice.

Any references in this information to non-IBM Web sites are provided for convenience only and do not in any manner serve as an endorsement of those Web sites. The materials at those Web sites are not part of the materials for this IBM product and use of those Web sites is at your own risk.

IBM may use or distribute any of the information you supply in any way it believes appropriate without incurring any obligation to you.

Information concerning non-IBM products was obtained from the suppliers of those products, their published announcements or other publicly available sources. IBM has not tested those products and cannot confirm the accuracy of performance, compatibility or any other claims related to non-IBM products. Questions on the capabilities of non-IBM products should be addressed to the suppliers of those products.

This information contains examples of data and reports used in daily business operations. To illustrate them as completely as possible, the examples include the names of individuals, companies, brands, and products. All of these names are fictitious and any similarity to the names and addresses used by an actual business enterprise is entirely coincidental.

COPYRIGHT LICENSE:
This information contains sample application programs in source language, which illustrates programming techniques on various operating platforms. You may copy, modify, and distribute these sample programs in any form without payment to IBM, for the purposes of developing, using, marketing or distributing application programs conforming to the application programming interface for the operating platform for which the sample programs are written. These examples have not been thoroughly tested under all conditions. IBM, therefore, cannot guarantee or imply reliability, serviceability, or function of these programs. You may copy, modify, and distribute these sample programs in any form without payment to IBM for the purposes of developing, using, marketing, or distributing application programs conforming to IBM's application programming interfaces.

Trademarks

The following terms are trademarks of the International Business Machines Corporation in the United States, other countries, or both:

@server®	AD/Cycle®	MQSeries®
@server®	AIX 5L™	OS/2®
Redbooks (logo) ™	AIX®	Redbooks™
developerWorks®	BookMaster®	Summit®
ibm.com®	Cloudscape™	SystemView®
pSeries®	CICS®	Tivoli®
xSeries®	IBM®	WebSphere®
zSeries®	LoadLeveler®	

The following terms are trademarks of International Business Machines Corporation and Rational Software Corporation, in the United States, other countries or both:

Rational Rose®	Rational®

The following terms are trademarks of other companies:

Microsoft, Windows, Windows NT, and the Windows logo are trademarks of Microsoft Corporation in the United States, other countries, or both.

Java and all Java-based trademarks and logos are trademarks or registered trademarks of Sun Microsystems, Inc. in the United States, other countries, or both.

UNIX is a registered trademark of The Open Group in the United States and other countries.

Other company, product, and service names may be trademarks or service marks of others.

Preface

This IBM® IRedbook is the fourth in a planned series of Redbooks™ publications addressing grid computing. In the first publication, *Introduction to Grid Computing with Globus*, SG24-6895, grid concepts and the Globus Toolkit were introduced. The second publication, *Enabling Applications for Grid Computing with Globus*, SG24-6936, introduced the concept of enabling applications to run on the open source Globus Toolkit 2.0. The third, *Globus Toolkit 3.0 Quick Start*, REDP3697, provides the critical jump start for someone who wants to learn about Globus Toolkit 3.0 but has little or no experience with prior Globus releases or grid computing in general.

The goal of this redbook is to familiarize the user with the concepts of the OGSA (Open Grid Services Architecture), OGSI (Open Grid Services Infrastructure), Globus Toolkit 3.0, presenting concrete programmatic examples, and also introducing the enhanced features of the IBM Grid Toolbox. We illustrate the various steps needed to develop a grid service application. Existing applications can be wrappered and made available as grid services or applications can be developed from scratch to take advantage of the grid service concepts and provide the benefits made possible by the grid service.

The redbook is organized into eight chapters:

- Chapter 1, “Introduction” on page 1

  This chapter summarizes the motivations and benefits of developing IT business solutions using grid computing technology and presents an overview of the main standards and organizations.

- Chapter 2, “Service Oriented Architecture” on page 5

  This chapter introduces the Service Oriented Architecture (SOA) as a technology to enable grid services, primarily making use of Web services.

- Chapter 3, “Open Grid Services Architecture” on page 21

  This chapter introduces the fundamentals of grid services and the challenges that they impose. Theses concepts, plus a general grid framework, form the Open Grid Services Architecture (OGSA), which uses Web Services as the main technology to enable grid services.

- Chapter 4, “Grid services development” on page 37

  This chapter describes a simplified method that embraces the complete development cycle of a grid service, providing straightforward guidelines on how to code, build and deploy a grid service in an arbitrary hosting environment.

- Chapter 5, “Major features of grid services” on page 59

  This chapter introduces the OGSI and GT3 features of grid services and show how these grid service features address the shortcomings of the Web Services model.

- Chapter 6, “Project and design of grid applications” on page 73

  This chapter provides an overview of the issues to consider for any grid application. The approach to building a grid-enabled application encompasses a wide range of aspects of problem analysis, application architecture, and design.

- Chapter 7, “Case study: grid application enablement” on page 115

  This chapter presents a case study which aims to apply the main concepts of grid service application design, specifying and coding over a real, practical grid application. The study project is an application of a bulletin service, as could be implemented by a news organization’s Web site.

- Chapter 8, “IBM Grid Toolbox basics” on page 177

  This chapter introduces the IBM Grid Toolbox V3 for Multiplatforms V1.1, the IBM implementation of the OGSI 1.0 specification.

The team that wrote this redbook

This redbook was produced by a team of specialists from around the world working at the International Technical Support Organization, Austin Center.

Luis Ferreira, also known as “Luix”, is a Senior Software Engineer at IBM Corporation - International Technical Support Organization, Austin Center, working on Linux and grid computing projects. He has 20 years of experience with UNIX®-like operating systems in design, architecture and implementation, and holds an M.S. degree in Systems Engineering from Universidade Federal do Rio de Janeiro in Brazil. Before joining the ITSO, Luis worked at Tivoli® Systems as a Certified Tivoli Consultant, at IBM Brasil as a Certified IT Specialist, and at Cobra Computadores as a kernel developer and operating systems designer.

Arun Thakore is a Certified Consulting I/T Architect at IBM, presently working at the AIS division, Architecture and Technology Center of Excellence Practice. He is also a member of the IBM Architect Certification Board. He has led engagements in grid and various e-business architectures in B2C and B2B areas. He has worked on the development of the reference architecture for the enablement of participants in an e-marketplace and on the reference architecture for e-business applications. He has also worked on the the course “Development of Operational Architecture Model” that is offered via distance learning. He has technically assisted in the pre-sales activities as well. Prior to his current

assignment, he worked as a lead Architect at the Distributed Reservation Systems department of the Travel and Transportation Solution unit at the BIS division. He has been involved in the architecting of distributed reservation systems for different customers. He has developed architectures for enabling the reservation system functionality to be accessed via a Web-based environment. He has provided technical guidance and leadership to the development team at the Travel and Transportation Solution Unit. In the past, he worked as an architect at the IBM Manufacturing Solution Unit, architecting various aspects of the solution for a manufacturing plant. He has also designed and led development teams in the development of distributed object servers. Arun has designed applications for small hand-held devices, developed architectures for integrating data from heterogeneous data sources in a medical environment, and developed application and data architectures for data assimilation applications for oceanic/atmospheric studies. He has experience in performance analysis and development of system management monitors and operational tasks. He has also trained software engineers in object-oriented software development methodology, relational database design and implementation, operating system concepts, and the design of applications using object-oriented databases. He holds a PhD with majors in Computer Science and Engineering and has published in various reputed journals and conferences in the field of information technology.

Michael Brown is a Senior Programmer and Sun-certified J2EE architect working in the IBM Linux Integration Center in Austin, Texas, where he is the leader of the team working on projects in the Americas. He has more than 25 years of experience as an application developer and enterprise architect on multiple platforms and operating systems, including UNIX, Linux, AIX®, and OS/2®. Michael holds B.S. (Honours) and M.S. degrees in Computer Science from the University of Western Ontario in London, Ontario, Canada. He worked on the previous GT3 Redpaper and presented GT3 programming sessions at the Colorado Software Summit®.

Fabiano Lucchese is currently the project manager of the grid computing team of Progonos Consulting. In 1994, Fabiano was admitted to the Computer Engineering undergraduate course of the State University of Campinas, Brazil and in mid-1997, he moved to France to finish his undergraduate studies at the Central School of Lyon. Also in France, he pursued graduate-level studies in Industrial Automation and, back in Brazil, joined Unisoma Mathematics for Productivity, where he worked as a software engineer on the development of image processing and optimization systems. From 2000 to 2002, he majored in Computer Engineering to achieve an M.S. from the State University of Campinas where he developed a task scheduling algorithm for balancing processing loads on heterogeneous grids.

Huang RuoBo is a staff software engineer in IBM China Software Development Lab, working with grid computing, grid services, Web services, and J2EE. He has more than three years of experience in Java™ development and is currently working in grid computing support and development in China.

Linda Lin is an advisory IT specialist in grid computing, IBM Server Group, located in China, working within Technical Sales Support. She has been with IBM for five years. She holds an M.S. in Computer Science from Beijing University of Aeronautics and Astronautics, China. Before joining the grid computing team, she worked in the IBM China Research Lab focused on Pervasive Computing and Wireless.

Paul Manesco is an IT specialist working in the Grid Design Center for eBusiness on Demand, IBM Server Group, located in Montpellier, France. He has been with IBM for five years and has been involved with grid computing since 2002, working on projects around the Globus Toolkit. He holds an M.S. in Computer Science from Universite de Montpellier, France. His areas of expertise include the Linux operation system, grid technologies and performance benchmarking on IBM xSeries® platforms.

Jeff Mausolf is a certified IT Architect in the IGS e-Technology Center in Austin, Texas. Jeff's current focus is on the grid computing initiative, where he is a member of an elite team working to integrate the grid with IBM's on-demand vision. Prior to joining the grid "brain-trust," Jeff worked as an application architect and software engineer on e-business engagements, where he developed portals for many state governments and agencies. He holds an M.S. in Computer Science and has been with IBM for twelve years. Before coming to IBM, Jeff was in the AeroSpace industry and held positions with Lockheed, Loral, and Ford AeroSpace, supporting contracts with the NASA at the Johnson Space Center in Houston, Texas. While at the Johnson Space Center, Jeff supported mission training for astronauts in the Shuttle Engineering Simulation (SES) laboratory,; he also helped to build Space Station and Mission Control training facilities, worked on the AP101S General Purpose Computer (GPC) for the Space Shuttle, and developed prototype Data Management Systems for the Space Station.

Karthik Subbian is a software engineer working for IBM Global Services, India. His primary expertise is application design, re-engineering and development for Telecom Business under UNIX platforms. He also has prior experience in assisting clients in managing their applications and migrating them to IBM platforms. His areas of interests in the software arena include grid computing, XML, SOAP and Web services. He holds a bachelor's degree in Electronics and Communication Engineering from Pondicherry Engineering College, India.

Nasser Momtaheni is a Senior Technology Consultant and a Certified IT Specialist, working in the IBM Solution Partnership Center in San Mateo,

California. He has been with IBM since 1991. Currently, as a member of the SPC grid team, he is responsible for grid application enablement on the Solutions Grid for Business Partners, equipped with both open and proprietary grid tools and solutions. He has over twenty years of experience in application development in distributed environments on multiple UNIX, AIX, and Linux platforms. In 2000, he joined the San Mateo Solution Partnership Center as a technical lead for the project Monterey, and led ISV efforts in enabling applications for AIX 5L™ on Itanium processors. He then led Linux application enablements on IBM hardware and software platforms for Independent Software vendors at the World Wide SPCs. He holds a BS degree in Business Management and Accounting, an M.S. degree in Mathematics, and an MS degree in Computer Sciences from the University of North Texas.

Olegario Hernandez is a former IBM Advisory Systems Engineer with 30 years of experience with IBM. He graduated as a Chemical Civil Engineer from the Universidad de Chile. During his time at IBM, he worked in application development, disciplines of systems management, IS Architecture, CICS® Application Interface, and business systems planning methodology (BSP). He has been assigned to residencies at different centers of the IBM International Technical Support Organization: ITSC Boeblingen for CICS Application Interface, ISC Gaithersburg for AD/Cycle®, and ITSC Poughkeepsie for Systems Management and SystemView®. After his retirement from IBM, he was a resident in ITSO Austin for the projects *Architecting Secure Systems with Tivoli products* and *Introduction to Grid Computing with Globus* as an IT Systems Consultant for IBM Business Partners.

Figure 1 The team that wrote this redbook (left to right): Michael, Paul, Arun, Luis, Jeff. On the bottom, left to right, are Nasser, Huang , Fabiano, Linda

Acknowledgements

Thanks to the following people for their contributions to this project:

Joanne Luedtke, Bart Jacob, Lupe Brown, Arzu Gucer, Wade Wallace, Chris Blatchley
International Technical Support Organization, Austin Center, USA

Cecilia Bardy
International Technical Support Organization, Raleigh Center, USA

John Adams
Grid Computing Initiative, e-Technology Center, IBM Austin, USA

Paul Magnone,
IGS EBO Business Development, IBM Somers, USA

Al Hamid
Architecture & Technology Center of Excellence, IBM Global Services, USA

John Caldwell
Enterprise Architecture & Technology, IBM Global Services, USA

Atul Kumar, Frank Paxhia, Mike Harris
Advanced Systems Infrastructure Development, IBM Poughkeepsie, USA

Ruth Harada and Flavio Carazato
Software Group, IBM Brazil

David A. Kra
Grid Computing Business Unit, IBM USA

Stephen Chu
Executive, Lead team, ISG, GCG, IBM China

Zhu QingJiu
China Software Development Lab, IBM China, USA

Globus Alliance team
Argonne National Laboratory, USA

Borja Sotomayor
BorjaNet and Universidad de Deusto, Spain

Eduardo J. Huerta
Progonos Consulting, Brazil

Joao Meidanis, Marco Aurelio Amaral Henriques
Unicamp - Universidade Estadual de Campinas, Brazil

Rogerio Luiz Iope, Luis Gustavo Gasparini Kiatake
USP - Universidade de São Paulo, Brazil

Special thanks to the following people:

The IBM's Grid Computing team in Somers, in particular to Tony White.

Become a published author

Join us for a two- to six-week residency program! Help write an IBM redbook dealing with specific products or solutions, while getting hands-on experience with leading-edge technologies. You'll team with IBM technical professionals, Business Partners and/or customers.

Your efforts will help increase product acceptance and customer satisfaction. As a bonus, you'll develop a network of contacts in IBM development labs, and increase your productivity and marketability.

Find out more about the residency program, browse the residency index, and apply online at:

ibm.com/redbooks/residencies.html

Comments welcome

Your comments are important to us!

We want our Redbooks to be as helpful as possible. Send us your comments about this or other Redbooks in one of the following ways:

- Use the online **Contact us** review redbook form found at:

  ibm.com/redbooks

- Send your comments in an Internet note to:

  redbook@us.ibm.com

- Mail your comments to:

  IBM Corporation, International Technical Support Organization
  Dept. JN9B Building 003 Internal Zip 2834
  11400 Burnet Road
  Austin, Texas 78758-3493

1

Introduction

This chapter summarizes the motivations and benefits of developing IT business solutions using grid computing technology, and presents an overview of the main standards and organizations.

1.1 Why grid computing?

Businesses are constantly faced with pressure to reduce the time to market of their product and services. In order to stay competitive, they are faced with the challenge of determining the right customer set and targeting their marketing activities to that customer set. Businesses increasingly have to work collaboratively with a variety of partners, suppliers, and various parts of their own organization in order to produce and supply products and services. These collaborations are getting complex and business needs demand shorter cycle times for these collaborative business processes.

The IT landscape of the various partners, suppliers, and other entities that a business deals with is characterized by application sets which use different and heterogeneous hardware platforms and operating environments, and are written using different programming models and languages. Businesses typically have silos of heterogeneous hardware and software resources within different parts of their own organization. Further, in order to achieve reliability, fault tolerance, and availability within the silos, the systems are designed for peak demand and are under-utilized most of the time. The time and accuracy of the various computing- and data-intensive tasks are limited by the availability of the resources within the silos.

1.2 Benefits of grid computing

Businesses can have a great advantage in meeting their challenges if they execute applications on a computer grid environment. Grid computing is an evolutionary step in distributed computing. Grid computing allows a pool of heterogeneous resources both within and outside of an organization to be virtualized and form a large, virtual computer. This virtual computer can be used by a collection of users and/or organizations in collaboration to solve their problems. The rules governing the participants providing the resources and the consumers using the resources, as well as the conditions for sharing, dictate the nature of the virtual organization of users of this virtual computer.

This use of a virtual computer formed from a grid of shared resources allows the users of the virtual organizations to solve complex collaborative problems. The sharing enabled by grid computing goes far beyond sharing through the exchange of data and permits the direct use of computing, data, and network resources. This enhanced sharing allows users and organizations to improve performance, reduce cycle time, increase availability, and improve fault tolerance by distributing both computing- and data-intensive workloads across several resources forming the grid. Grid computing also enables the effective and efficient integration of applications that are on different and heterogeneous

hardware platforms, operating environments, and written using different programming models and languages, to implement collaborative business processes. Further, the sharing of the resources from the individual silos in forming the bigger pool allows the under-utilized and available resources to be discovered and used by applications and jobs that have a greater need. This not only improves the utilization of underused resources, but also improves the performance and availability of resource-intensive jobs and applications.

1.3 Use of standards

The vision of grid computing can be implemented by the use of open standards with common interfaces and protocols for defining, discovering and using the heterogeneous set of resources.

GGF and Globus alliance

The Global Grid Forum (GGF) is an organization that has undertaken the mission of creating and documenting the technical specifications and implementation guidelines that promote and support the development, deployment, and implementation of grid technologies and applications. Specifically, GGF has published a standard Open Grid Services Architecture (OGSA), and an Open Grid Services Infrastructure (OGSI 1.0) based on OGSA.

A reference implementation of OGSA called Globus Toolkit 3.0 (GT3), developed by Globus Alliance (http://www.globus.org/), has also been made available to assist the grid computing community. The grid framework published at GGF is itself based on the Service Oriented Architecture (SOA), the Web services, and Internet standards from the World Wide Web Consortium (W3C). IBM is an active participant of the GGF and has contributed to the development of the standards. In addition, IBM has developed a version of the grid toolkit, called the IBM Grid Toolbox V3 for Multiplatforms V1.1, which follows the OGSA architecture and the OGSI specification. IBM Grid Toolbox enhances the features provided by GT3 and also makes it easier for the applications to be deployed in industrial strength application server environments.

Web Services Resource Framework: the new specification

The GGF's OGSI- Working Group issued a new draft proposal WS-Resource Framework (WSRF) as an OGSI 1.0 evolution in January 2004. Syntax and terminology have changed, and the specification was broken up into separate specifications, each focusing on a particular area. The document *From Open Grid Services Infrastructure to WS-Resource Framework: Refactoring & Evolution Version 1.0 version 1.0*, from 2/12/2004 introduces the following normative WSRF specifications. More information about WSRF can be found in Appendix G, “WSRF” on page 261.

Important: All examples of this document are based on the OGSI 1.0 specification.

2

Service Oriented Architecture

This chapter introduces the Service Oriented Architecture (SOA) as a technology to enable grid services, primarily making use of Web services.

In order to make this possible, complementary specifications to Web services are presented, including: Web Services Description Language (WSDL) for describing the content and usage of the Web services; the Simple Object Access Protocol standard (SOAP) as a protocol for exchanging messages between Web services; and the Universal Description, Discovery and Integration (UDDI) specification for allowing services publication and discovery.

The basics concepts and their main elements are presented, each followed by practical examples.

2.1 What is SOA?

The Service Oriented Architecture (SOA) is an architectural approach whereby an application is composed of independent, distributed and co-operating components called *services*. This collection of services constitutes the application. The services can be distributed within or outside of the organizational physical boundaries and security domains. Furthermore, the various service components can exist on varying platforms and can be implemented using different programming languages.

The key concept of SOA is that the functionality implemented by a service is exposed via a standard-based interface declaration. The implementation details are hidden from the users of the service; they invoke the service based on the operations exposed in these interfaces. One interesting way of implementing the SOA to take advantage of Web services, which can be used in the process of service definition, discovery and execution, will be discussed in the next sub-sections.

Furthermore, with the aid of other services such as resource schedulers, index services and discovery, the applications can be dynamically configured to take advantage of similar functions available for varied sources to deliver their functionality. This results in improved and predictable service levels and optimized utilization of resources.

2.2 The basic components of SOA

The SOA's basic components are elements and the operations messages they exchange with each other.

There are three key elements: Service Provider, Service Requestor and Service Registry, as shown in Figure 2-1 on page 7.

Service Provider	The Service Provider is responsible for building a useful service, creating a service description for it, publishing that service description to one or more service registries, and receiving service invocation messages from one or more Service Requestors.
Service Requestor	The Service Requestor is responsible for finding a service description published to one or more Service Registries, such as yellow pages for services, and for using service descriptions to bind to or invoke services hosted by Service Providers. Any consumer of a service can be considered a Service Requestor.

Service Registry — The Service Registry is responsible for advertising service descriptions published to it by the Service Providers, and for allowing Service Requestors to search the collection of service descriptions contained within the Service Registry. Once the Service Registry provides a match between the Service Requestor and the Service Provider, the Service Registry is no longer needed for the interaction.

An application component can play any of the above roles. Note also that an application component can play more than one role. For example, an application component that performs Order Processing can be implemented as a Service Provider and will allow client modules to invoke this service, taking order input from the users. In this case, the client module plays the role of the Service Requestor. However, the Order Processing service can also invoke the services offered by a Credit Authorization service before accepting and processing the orders. Therefore, the Order Processing service plays the role of both Service Provider and Service Requestor.

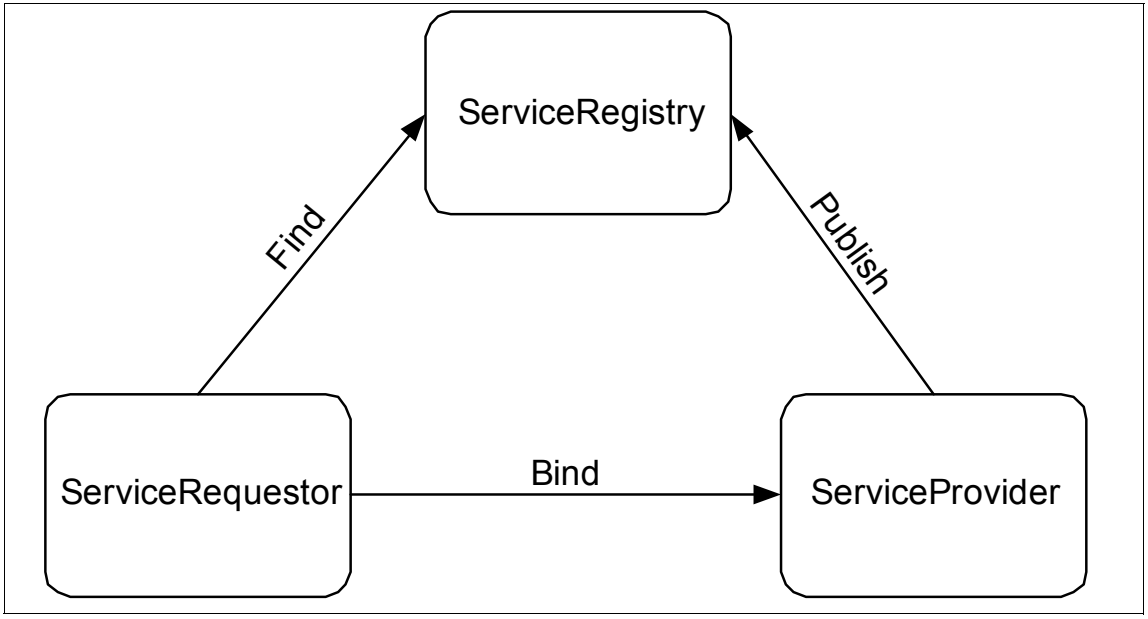


Figure 2-1 Elements of the Service Oriented Architecture (SOA)

Operations are defined by contracts between the above elements. There are also three: Publish, Find and Bind, as shown in Figure 2-1.

Publish — The Publish operation is a contract between the Service Provider and the Service Registry. The Service Provider registers the services interfaces it provides at the Service

	Registry using the Publish operation. Once published, the services provided by the Service Provider are available for any Service Requestor to use.
Find	The Find operation is a contract between the Service Requestor and Service Registry. The Service Requestor uses the Find operation to get a list of the Service Providers that satisfies its needs. It may indicate one or more search criteria, such as the desired availability and performance, in the Find operation. The Service Registry searches through all the registered Service Providers and returns the appropriate information.
Bind	The Bind operation is a contract between the Service Requestor and the Service Provider. It allows the Service Requestor to connect to the Service Provider before invoking the operations. It also enables the Service Requestor to generate the client-side proxy for the service provided by the Service Provider. The binding can be dynamic or static: in the first case, the Service Requestor generates the client-side proxy based on the service description obtained from the Service Registry at the time the service is invoked; the other case involves the Service Requestor generating the client-side proxy during application development.

2.3 Web services as an implementation of the SOA

Web services are an emerging technology widely used for implementing the SOA. They employ a program-to-program communication model built on existing Internet standard eXtensible Markup Language (XML) for the specification of data in a platform, language, hardware delivery device, and software vendor neutral manner. Web services do not specify a particular protocol for communication, thus any communication layer protocol, such as HTTP or JMS, can be used in the message exchange process.

For the purposes of grid services, Web services utilize the Web Services Description Language (WSDL) to describe content and usage, the emerging standards of SOAP as a protocol for sending exchange messages between Web services, and the Universal Description, Discovery and Integration (UDDI) specification to allow Web providers to register their services and Web requestors to locate the appropriate services providers.

Additional standards for defining and implementing quality of service for Web services are being defined in the WS-Security, WS-Reliable Messaging,

WS-Coordination, and WS-Transaction families of specifications. Similarly, standards for utilizing Web services in the implementation of collaborative business processes are being defined in the WS-Business Process Execution Layer specification. Refer to *Web Services Conceptual Architecture*, May 2001, by Heather Kreger - IBM Software Group, at:

http://www.ibm.com/software/solutions/webservices/pdf/WSCA.pdf

2.3.1 Web Service Description Language (WSDL)

Web Service Description Language (WSDL) is an XML-based standard from the World Wide Web Consortium (W3C) for describing the interface and usage bindings for Web services. Since it is XML-based, it is independent of programming languages and development environments.

It is used by Service Providers to describe the particularities of the services which are published, through UDDI specification, in the service registries. Thus, Service Requestors are able to search for the proper Service Provider and invoke the desired service based on the information expressed in WSDL.

A WSDL document contains three categories of information about the Web service: Service Interface, Service Bindings, and Service Implementation. The Service Interface defines the structure of the data communicated and the signature of the operations provided by the service in a language, platform, and communication protocol independent fashion. The service binding specifies the transport protocols to be used and the encoding rules to be followed when accessing the public operations provided by the service. The Service Implementation specifies the details of the implementation for all operations of the service.

In the next few sections, each of these categories is explored with references to appropriate examples.

Service Interface

The key elements of the Service Interface are Types, Message, Operation and Port Type, discussed here with the example provided in Figure 2-2 on page 10.

Types The data type definitions used by the messages exchanged between the Service Requestor and the Service Provider are specified in the Types section. In the example, there is one complex type named Arrayof_xsd_string which describes an array of strings. As shown in the next paragraph, this type will be used in the description of the messages.

```
<?xml version="1.0" encoding="UTF-8"?>
<wsdl:definitions targetNamespace="http://DefaultNamespace"
   xmlns:impl=http://DefaultNamespace
   xmlns:intf="http://DefaultNamespace"
   xmlns:apachesoap="http://xml.apache.org/xml-soap"
   xmlns:wsdlsoap="http://schemas.xmlsoap.org/wsdl/soap/"
   xmlns:soapenc="http://schemas.xmlsoap.org/soap/encoding/"
   xmlns:xsd="http://www.w3.org/2001/XMLSchema"
   xmlns:wsdl="http://schemas.xmlsoap.org/wsdl/"
   xmlns="http://schemas.xmlsoap.org/wsdl/">

 <wsdl:types>
   <schema xmlns="http://www.w3.org/2001/XMLSchema"
       targetNamespace="http://DefaultNamespace">
     <import namespace="http://schemas.xmlsoap.org/soap/encoding/"/>
     <complexType name="ArrayOf_xsd_string">
       <complexContent>
         <restriction base="soapenc:Array">
           <attribute ref="soapenc:arrayType" wsdl:arrayType="xsd:string[]"/>
         </restriction>
       </complexContent>
     </complexType>
   </schema>
 </wsdl:types>

 <wsdl:message name="getMOTDResponse">
   <wsdl:part name="getMOTDReturn" type="xsd:string"/>
 </wsdl:message>

 <wsdl:message name="getMOTDRequest">
   <wsdl:part name="in0" type="impl:ArrayOf_xsd_string"/>
 </wsdl:message>

 <wsdl:portType name="MOTD1">
   <wsdl:operation name="getMOTD" parameterOrder="in0">
     <wsdl:input name="getMOTDRequest" message="impl:getMOTDRequest"/>
     <wsdl:output name="getMOTDResponse" message="impl:getMOTDResponse"/>
   </wsdl:operation>
 </wsdl:portType>
```

Figure 2-2 Service Interface definition part of an example Web service definition

Messages

A message represents a single interaction between the Service Requestor and Service Provider. If an operation is a Remote Procedure Call (RPC) that requests a return value, then the interaction is bi-directional and has to be defined using two messages. In the example, two

messages are defined: getMOTDRequest and getMOTDResponse. The getMOTDRequest message is sent from the Service Requestor to the Service Provider. The Service Provider responds by sending the getMOTDResponse message to the Service Requestor. A message can have one or more typed parts. In the example, the getMOTDRequest consists of one part of type Arrayof_xsd_string defined earlier in the WSDL. Similarly, the getMOTDResponse consists of one part of type string. These messages will be used in the definition of operations supported by the Web service, as shown in the next paragraphs.

Operation

An operation is a description of an action supported by the Web service. Operations can be of any of the four types of Message Exchange Patterns (MEP): One-way, Request-response, Solicit-response and Notification. Table 2-1 provides the order of message communication and the description of the MEPs. Using a One-way operation, the Service Requestor is only allowed to send a one-way input message to trigger the operation without receiving any immediate response from the Service Provider. In Request-response, the Service Provider responds with the result of the operation in the form of a response output message upon receiving an input trigger message. Solicit-response is used if the Service Provider solicits a response from the Service Requestor by sending a message, and the Service Requestor responds with a response. Using Notification, the Service Provider sends a one-way notification message to the Service Requestor. In the example, the getMOTD operation is a Request-response. The getMOTDRequest message is the input of the operation and the getMOTDResponse is the output response of the operation.

Table 2-1 Message Exchange Patterns (MEP) used by the operation

MEP	Order	Description
One-way	Input element	The endpoint receives a message
Request-response	Input element Output element	The endpoint receives a message and sends a correlated message

MEP	Order	Description
Solicit-response	Output element Input element	The endpoint sends a message and receives a correlated message
Notification	Output element	The endpoint sends a message

Port type A port type (or PortType) is a collection of operations that are supported by the Web service. Port types are similar to interfaces in Java. In the example, there is one port type, namely MOTD1. It consists of one operation called getMOTD.

Bindings

The binding specifies the details about the use of the transport protocol for the transmission between the Service Requestor and the Service Provider for a given port type. A service can support multiple bindings for a given port type. Each binding is addressed via a unique Uniform Resource Identifier (URI). Figure 2-3 on page 13 shows the relationship between a service, its port type, bindings, their URIs and the messages that are transferred. As can be seen, a service provides access to a given resource. It may contains multiple port types. Each port type defines one or more operations. More than one binding can be connected to one port type.

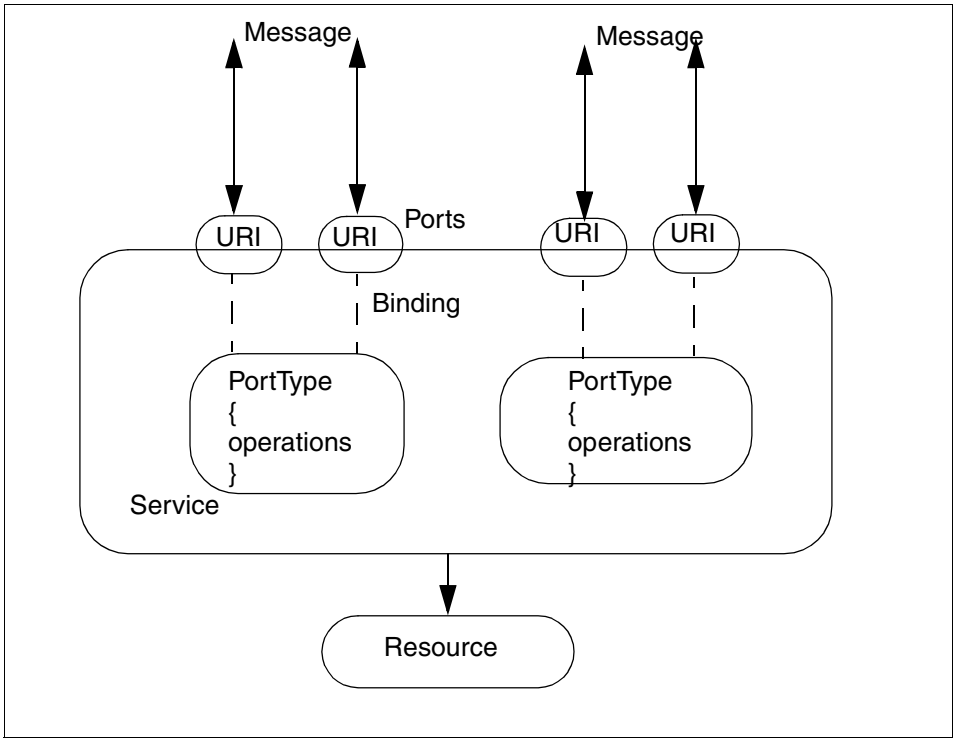


Figure 2-3 Relationship of WSDL elements

Figure 2-4 shows an example of bindings. In a WSDL document, the binding information is specified within the binding section and contains the following elements.

```
<wsdl:binding name="MOTDSoapBinding" type="impl:MOTD1">
  <wsdlsoap:binding style="rpc" transport="http://schemas.xmlsoap.org/soap/http"/>
  <wsdl:operation name="getMOTD">
    <wsdlsoap:operation soapAction=""/>
    <wsdl:input name="getMOTDRequest">
      <wsdlsoap:body use="encoded"
         encodingStyle="http://schemas.xmlsoap.org/soap/encoding/"
         namespace="http://DefaultNamespace"/>
    </wsdl:input>
    <wsdl:output name="getMOTDResponse">
      <wsdlsoap:body use="encoded"
         encodingStyle="http://schemas.xmlsoap.org/soap/encoding/"
         namespace="http://DefaultNamespace"/>
    </wsdl:output>
  </wsdl:operation>
</wsdl:binding>
```

Figure 2-4 Service bindings definition part of an example Web service definition

Name and type The binding is specified for a given port type. In the example, the binding name is MOTDSoapBinding and it is specified for the MOTD1 port type.

Style The binding style defines the communication used by the transport protocol when invoking the operations of the specified port type. The WSDL specification defines two types of binding styles for SOAP: Document style and RPC style. Using Document style, all the information is encapsulated in one XML document and is defined by an XML schema. Using RPC style, the body of a SOAP message is used to represent a function call and the elements are represented as parameters within it. In the example, an RPC type of communication is supported.

Transport The transport specifies the protocol used for the communication. The transport details specified in the binding are applicable for all the operations defined within the specified port type. In the example, SOAP over HTTP was chosen.

Encoding style In addition to the binding style, the WSDL specification specify two encoding styles for each message set in the use attribute: Encoded or Literal. The Encoded style uses SOAP encoding rules to map the abstract data types to

concrete data. On the contrary, using Literal encoding the abstract data type produces the concrete data. Combining the binding style and encoding style provides four different communication models, namely, RPC/Encoded, RPC/Literal, Document/Encoded, and Document/Literal. In the example, the getMOTD operation with the associated getMOTDRequest and getMOTDResponse messages uses the RPC/Encoded style of communication. In addition, the rules for encoding and decoding the data are specified at the address http://schemas.xmlsoap.org/soap/encoding/. The types to be used are defined in the `http://DefaultNamespace` namespace.

Service Implementation

The Service Implementation provides implementation-specific details that the Service Requestor can use to request the various operations offered by the Web service. An example is shown in Figure 2-5.

```
<wsdl:service name="MOTD1Service">
  <wsdl:port name="MOTD" binding="impl:MOTDSoapBinding">
    <wsdlsoap:address location="http://localhost:8080/services/MOTD"/>
  </wsdl:port>
</wsdl:service>
```

Figure 2-5 Service implementation definition part of an example Web service definition

The following are the key elements specified for the Service Implementation:

Service The service name defines the name of the Web service that is providing the operations specified in the port types. In the example, the service name is MOTD1Service.

Port A port is merely an end-point that is offering the service. The Service Requestor binds or connects to the port to access the service. The name of the port, the protocol bindings that it will use for communicating and the address of the port when using the specified protocol are specified in the WSDL. In the example, the name of the port is MOTD; it uses the previous defined MOTDSoapBinding binding, and has an address of http://localhost:8080/services/MOTD.

2.3.2 Simple Object Access Protocol (SOAP)

Simple Object Access Protocol (SOAP) is an XML-based messaging protocol. The SOAP specification defines a mechanism for the exchange of structured information in a decentralized, distributed environment. Within the Web services framework, SOAP is used as a protocol for communication between the three key elements of SOA defined earlier: the Service Provider, Service Requestor, and Service Registry.

SOAP is platform- and language-independent and hence can be effectively used for communication between the Web service entities implemented in a variety of languages and across several platforms. It is also transport protocol independent. Hence, it can be used with a variety of transport protocols, even though its use with the HTTP protocol for Web services is very common.

In order to illustrate this communication, Figure 2-6 and Figure 2-7 on page 17 show a sample SOAP message sent from a Service Requestor to a Service Provider, requesting an invocation of an operation specified within the Web service, and its respective returned message. Both messages use the HTTP protocol for transport and are embedded in an HTTP request and response.

```
<?xml version="1.0" encoding="UTF-8"?>
<soapenv:Envelope xmlns:soapenv="http://schemas.xmlsoap.org/soap/envelope/"
     xmlns:xsd="http://www.w3.org/2001/XMLSchema"
     xmlns:xsi="http://www.w3.org/2001/XMLSchema-instance">
  <soapenv:Body>
    <ns1:getMOTD soapenv:encodingStyle="http://schemas.xmlsoap.org/soap/encoding/"
         xmlns:ns1="urn:itso">
      <ns1:arg0 xsi:type="xsd:string">world</ns1:arg0>
    </ns1:getMOTD>
  </soapenv:Body>
</soapenv:Envelope>
```

Figure 2-6 SOAP Message from the Service Requestor to the Service Provider

```
<?xml version="1.0" encoding="UTF-8"?>
<soapenv:Envelope xmlns:soapenv="http://schemas.xmlsoap.org/soap/envelope/"
     xmlns:xsd="http://www.w3.org/2001/XMLSchema"
     xmlns:xsi="http://www.w3.org/2001/XMLSchema-instance">
  <soapenv:Body>
    <ns1:getMOTDResponse soapenv:encodingStyle="http://schemas.xmlsoap.org/soap/encoding/"
         xmlns:ns1="urn:itso">
      <getMOTDReturn xsi:type="xsd:string">hello: world</getMOTDReturn>
    </ns1:getMOTDResponse>
  </soapenv:Body>
</soapenv:Envelope>
```

Figure 2-7 SOAP message from the Service Provider to the Service Requestor

The following SOAP elements shall be noted:

Table 2-2 SOAP elements

Element name	Function
Namespaces	In order to provide interoperability between different programming languages that can potentially implement and use the Web service, various language-independent namespaces are defined and specified within the SOAP message. As can be seen in the example, the schema that defines the elements of the SOAP envelope themselves are specified in the `xmlns:soapenv = http://schemas.xmlsoap.org/soap/envelope/` namespace. Similarly, `xmlns:xsd = "http://www.w3.org/2001/XMLSchema"` and `xmlns:xsi = "http://www.w3.org/2001/XMLSchema-instance"` specify the namespaces describing the schemas of the various elements used in the SOAP message.

Element name	Function
Encodings	Encodings define how data values defined in the application can be translated to and from the protocol format, here the format specified in the language-neutral SOAP messages. These translation steps are referred to as *serialization* and *deserialization*. Thus, the SOAP encoding tells the SOAP runtime environment how to translate from data structures constructed in a specific programming language into SOAP XML, and vice versa. The `soapenv:encodingStyle = "http://schemas.xmlsoap.org/soap/encoding/"` present in the SOAP message body specifies the encoding to be used for serializing and deserializing the data elements specified in the SOAP message.
Uniform Resource Name (URN) of the Web service	The URN uniquely specifies the Web service to the Service Requestor. The URN must be unique among all the services deployed in the SOAP server, that is the implementation engine of the Web services provider. The URN in the example is `itso` and it is specified as `urn:itso`. This URN is unique within the SOAP server `127.0.0.1/axis/services/MOTD` specified in the transport protocol dependent HTTP header.
Method name	The name of the operation or the method being invoked within the Web service. The method name is specified in the SOAP body. In the example of Figure 2-7 **"**`ns1:getMOTD`**"** is the specification of the method getMOTD within the Web service.
Input parameter	The method may take as input zero or more parameters. The parameters are specified in the SOAP body right after the definition of the method name. In the example shown in Figure 2-6, the method has one input argument and is specified with the `ns1:arg0` tag. It takes a string argument and a value of `world` is passed.

Element name	Function
Output result	The method can potentially return an output value as a result of the invocation. The output value is returned from the Service Provider to the Service Requestor in a response SOAP message. Figure 2-7 shows the response message; the output result value is specified within the getMOTDReturn tags. A string value of `hello world` is returned.

2.3.3 Universal Description, Discovery, and Integration (UDDI)

Universal Description, Discovery and Integration (UDDI) is a specification that defines the mechanisms for storing and searching the Web services definitions defined in the WSDL documents. The Service Provider advertises the Web services it offers by registering the WSDL definitions of its services in the UDDI registry. The Service Requestor obtains the service details from the UDDI registry either at build time or at runtime. There are tools available on the market that enable the Service Requestor to generate language-, platform- and protocol-specific service proxies based on the service information provided in the WSDL document. Hence, at build time, the Service Requestor can search for the appropriate WSDL, generate the proxy, and incorporate it in its application to access the service offered by the Web service.

In addition to the tools, there are language-specific libraries available for programmatic interaction with the UDDI registry. For example, UDDI4J, supported by IBM, is a Java class library for interacting with the UDDI registry. Similarly, there are language-specific packages to dynamically interpret the WSDLs and to invoke the Web services remotely. Using these runtime libraries, the Service Provider can obtain the service descriptions from the UDDI registry dynamically and invoke the services at runtime.

3

Open Grid Services Architecture

This chapter introduces the fundamentals of grid services and the challenges that they impose. Theses concepts, plus a general grid framework form the Open Grid Services Architecture (OGSA), which uses Web services as its main technology to enable grid services.

3.1 Introduction

The OGSA concepts drive the development of the interfaces and protocol specifications, called Open Grid Services Infrastructure (OGSI), which extends WSDL and XML specifications and provides other mechanisms to satisfy grid requirements, and is also shown in this chapter, in addition to a brief overview of the incoming evolutions.

The major goal of grid technology is to promote a virtual computing environment, which means the use of a set of services, through the transparent and coordinated use of distributed and heterogeneous resources. Services are the abstraction of resources, and they can be computing cycles, software, documents, data, storage, and so on.

In order to make this both real and useful, some challenges must be met:

- *Transparent* means that the user will use this environment with the same quality he or she would get using a local system. Therefore, the grid shall provide easy tools to assist users to specify their needs of services and qualities and good Quality of Service (QoS), which in this case means fast services access using smart authentication and high-speed communication.
- *Coordinate* means that it is necessary to have a management system that provides matching users' needs and resource availability, monitoring the services' use and providing additional services facilities, such as local resource control, resource performance and state control, logging, and security, among others.
- *Distributed and heterogeneous* means that interoperability is necessary, and so is a common and standardized interface that translates users' needs and resources availability in one single language, regardless of the hardware, software and operation system of each distributed resource.

One can observe that the use of Web services, and further the Web Service Description Language (WSDL), the Simple Object Access Protocol (SOAP) and the Universal Description, Discovery and Integration (UDDI), match grid services requirements. These specifications provide much of the interoperability and transparence in grid services demand.

As can be seen in this scenario, the Globus Alliance has developed the first implementation Open Grid Services Architecture (OGSA), and then takes into account the contributions of the Global Grid Forum's OGSA Working Group (GGF-OGSA-WG).

The main idea of OGSA is to define the framework, architecture and functionalities of grid systems that will drive the development of the Open Grid Services Infrastructure (OGSI) by the GGF-OGSI-WG.

OGSI defines the conventions and specifications of the grid services, their protocols and interfaces' behavior, properties and attributes, based on Web services specifications plus many other extensions. This document is based on the OGSI version 1.0 because it is a version that is more known and ready to implement, more suitable for commercial applications. However, evolutions are been considered by OGSA-WG, such as that issued in January 2004, a first version of the Web Services Resource Framework (WSRF).

With OGSI, it is already possible to implement grid applications. However, there are some toolkits available which provide libraries and tools to aid this process, like the Globus Toolkit, currently in version 3 (GT3), and the IBM Grid Toolbox. These will be presented in the next chapters.

Figure 3-1 summarize the role scenario and the main technologies involved in the deployment of grid services: OGSA, OGSI and GT3.

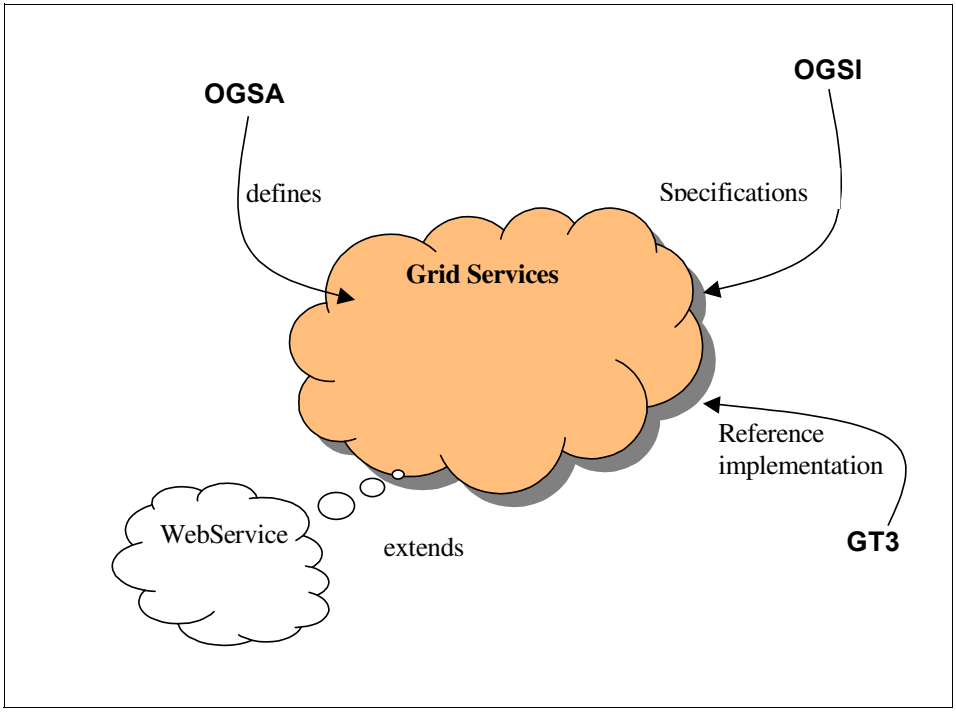


Figure 3-1 The relationship among OGSA, OGSI and GT3

3.2 OGSA mechanisms

This chapter presents some issues and directions taken by Globus and GGF in order to address the challenges of enabling virtual organizations. Some of them are discussed in detail in Chapter 5, “Major features of grid services” on page 59.

3.2.1 Interoperability

The problem of seamless integration of services architected on OGSA was addressed by separating the interfaces definition and the protocol bindings through WSDL and the usage of Service Oriented Architecture for OGSA framework.

The idea of uniform service interface definition (that is, service semantics) includes a layer of abstraction to hide the platform specific implementation. This adds up to the concept of virtualization of services and adds to the fluidity of virtual organization.

The quality of service (reliability), authentication, authorization, credential delegation are handled as the properties of bindings.

The advantage of this implementation is more fluidity in forming various groups of interfaces to protocol bindings as required by the client. Also, the abstraction provided by the grid service interface will provide more implementation-free integration of client services to the services of a service provider. The freedom of implementation of services based on native platform facilities and the global rule for grid services to abide by uniform grid service interface pattern thus solve the problem of interoperability.

3.2.2 Discovery and access of resources

The requirement of discovering and accessing the resources was addressed in three parts in OGSA:

1. A standard representation of service data, containing information about grid service instance and represented in XML structures.
2. A standard operation, FindServiceData, to retrieve service data from individual grid service instances.
3. A standard interface registry, for registering the information about the grid service instances.

3.2.3 Independent upgradability

The problem of upgradability of grid service instances and maintenance of versioning information was addressed by defining OGSA mechanisms, to refresh the client's knowledge of service, such as an upgrade of services supported, an upgrade to a host platform or any other domain specific upgrade details applicable to the client. The service description includes protocol-binding properties that will be used to communicate with the service. Two properties are often needed in such communications: reliable service invocation and authentication.

3.2.4 Transient life cycle management of resources

In a dynamic environment, services are created and need to be destroyed when no longer needed. The grid services architecture addresses this life cycle issue through a soft state approach, where grid services are created with a specified initial lifetime. The initial lifetime can be extended by a specified time period by explicit request of the client service or by another service, which has credential delegations of the client. In this architecture, client will be sending "Keep alive" messages to keep the service active in a providers system. If the client does not need the service, it stops sending "keep alive" messages. This situation can also be due to the breakdown of the software system at the client's end or any component failure in the grid services workflow. After the expiration of a set time period for the provider's service, the provider's host environment or the service by itself has the freedom to terminate and release the resources consumed so far for the purpose of client.

The grid service interface has an operation called SetTerminationTime for the purpose of setting up an initial time period for a service, used by the client. There are operations available for grid service instances, with which the client will be able to know when the service will terminate. Depending on computational needs, the extend operation can also be used to extend the lifetime of the desired services.

Also, the client can negotiate the expiration time set during the initial creation of the service using a set of operations for grid service. This operation allows the client to send maximum and minimum acceptable expiration times for a service. If the provider agrees to a customer request and can provide a service whose expiration time falls between the maximum and minimum, then the factory service instance creates the service for the client.

The clock synchronization used here for the process of determining time is based on Network Time Protocol (NTP). Using this protocol, time is synchronized between all grid services to an accuracy of tens of milliseconds.

3.2.5 Services state - grid service handle and reference

The state nature of every service actually changes throughout its lifetime, so it is necessary to have some process that manages these states. OGSA addresses this requirement by defining Service Data Elements that will store the service states and maintain them until the end of the service lifetime. The stored service state will be accessible via the grid service interfaces defined in OGSA. The change in service states will be passed on to co-services in the grid by asynchronous state change notification. Traditional messaging protocols techniques like publishing/subscription will be used.

The co-services interested in receiving the state changes will request notification from the service provider's service. Similarly, the provider's services interested in notifying all client services will publish their state-related information for the information of all co-services.

In the grid architecture, the service definition is separate from the actual service instances. One single interface can have more than one instance active at the same time, serving multiple client services. Hence, one of the OGSA services creates service instances handle and maps them to actual service interface definition handle. Therefore, a mapping between the service definition and the service instance is maintained in registry of OGSA.

As the service instance is created, a unique service instance identifier is allocated to it, called Grid Service Handle (GSH). This is invariable and unique to the service instance even over a time period. If a co-service wants to re-start the terminated service instance and regain control of some resources in the client system, it can do so using a GSH, since it remains unique even over time.

There is also more information about that specific service stored by the Grid Service Reference (GSR). In a different way than GSH, GSR may vary over time for a single service instance over the lifetime of the service and has a set termination time after which it expires. For example, the versioning information of the grid service and protocol binding information to the grid service are not carried by GSH; instead, GSR maintains it.

OGSA also defines mapping mechanisms for obtaining the updated GSR. The result of using an expired GSR is undefined.

3.2.6 Factory

OGSA defines an interface called Factory for creating new grid service instances. The Factory interface receives requests from client services and responds with a GSH and initial GSR after successfully creating the service instance. As complexity increases in the grid, from simple hosting environments to collective virtual hosting environments, factories will have multiple levels, for

instance higher and lower. Higher-level factories will delegate work to one or many of the multiple lower level factories under its reign, to accomplish a specific task.

3.2.7 Dynamic resolution of transient references from permanent handles

After the generation of GSH and initial GSR by the factory, OGSA defines a dynamic way to resolve the references between GSR and GSH through a new interface called HandleMap. This allows the client service to identify the new GSR. This interface maintains the latest mapping between the Handles (GSH) and References (GSR). The handle map interface will not return references to service instances that it knows have terminated. On the other hand, for the client service, possession of GSR does not imply it is valid, because by the time the client receives a GSR, it may have already expired or terminated for various reasons.

The handle map interface will return the most recent and valid GSR, if supplied a GSH. To identify the handle map interface, GSH will have the URL of the (home) HandleMap interface included in it. Thus, once GSH is obtained, GSR can be obtained by contacting the handle map interface.

But the protocol information to contact handle map information will be in GSR and not in GSH. How do we contact the handle map, and what protocol should be used for communication with HandleMap? The grid team decided to use the HTTP GET operation to speak to the HandleMap interface using the supplied GSH. In return, HandleMap returns GSR for the GSH requested in WSDL format.

Also, there must be a way in which Factory and HandleMap interfaces can communicate. Factory, as soon as it creates GSH (which contains the contact information and URL to contact the HandleMap interface), registers it to (home) HandleMap for a specific GSR. As the GSR evolves over a period of time, HandleMap maintains a track of it and supplies the recent WSDL document to the client services upon request.

3.2.8 Service data element and registry interface

Every service instance in grid technology has its own unique information needed later during its lifetime (such as time-to-live information of the service, GSH, GSR, HandleMap, etc.). All these data elements are nothing but XML elements contained in a single wrapper called Service Data Element.

Using the FindServiceData, a WSDL operation, one can retrieve the Service Data Elements for a particular service instance. But, for all this to happen,

Service Data Element needs to be stored and maintained in a specific place. Hence the need for Registry Interface, which provides the service of supplying Service Data Elements for a given service instance.

It is important to tie *Factory*, *Registry* and *HandleMap* since they are linked to each other in a particular fashion. When a factory receives a client request to create a grid service instance, the factory invokes hosting-environment specific capabilities to create the new instance, assigns it a handle (GSH), registers the instance with a service registry, and makes the handle available to the *HandleMap* service.

3.2.9 Asynchronous notification of state changes

The changes to the state of the service will take place throughout the existence of the service's lifetime. The co-services and services that are authorized to received and process information related to grid service state change will subscribe to the grid service for notification (source/sink). Similarly, the service provider service will use grid service interfaces for notification to publish the state change related information for other co-services which need them.

The OGSA framework addresses notification services in two parts. One is the Sender of notification, called the notification source, which will implement the notification source interface to publish and receive subscriptions to its notification messages. The second part is the receiver of notification messages, called the notification receiver, which will use the notification sink interface to receive the notification message.

The service that wishes to receive notification will supply the GSH (for the service, which needs notification) to the notification source interface. Then, notification messages start flowing to and from the notification source/sink. Reliability of the messages delivered is left to the protocol bindings of these services. These could be open protocols like UDP or proprietary messaging services.

3.3 Open Grid Services Infrastructure (OGSI)

The Open Grid Services Infrastructure 1.0 (OGSI) provides the conventions and specifications of many actions that take place in a grid system, for instance, the requester's services of creating, discovering and interacting, as well as other management services.

OGSI extends WSDL version 1.1 and XML definitions, improving new services in order to fit grid needs. Many of these extensions have been incorporated in WSDL 1.2. However, it is important to pay attention to the upcoming

specifications published by GGF, such as WSRF. More information about WSRF can be found in Appendix G, “WSRF” on page 261.

3.3.1 OGSI interfaces and their operations

The grid service is the base component of this distributed component object model, so-called OGSA, even though it is not a completely traditional distributed object-ased system. There are different port types (portType) that add on to this grid service (like a base class in the object model) to extend a combination of grid service port types, so as to meet different requirements we discussed in 3.2, “OGSA mechanisms” on page 24. Different port types that extend the grid service functionality will be discussed in detail in the next sections.

One extension proposed by OSGI to the Web services community is the mechanism of Service Data. Its role is to expose the instance of a service to requestors, providing a stateful system, through values of Service Data Elements (SDE).

We will start by explaining the GridService port type, which is the basic port type. We will discuss the purpose of each port type and operations that are available in each port type. As the definitions are highly evolving in this area, it is suggested that you refer to the GGF Web site (http://www.ggf.org/) for the latest version of Open Grid Services Infrastructure specifications.

GridService port type

We have stated that everything implemented in grid technology is a service and can be accessed via service operations. This could vary from notification to lifetime management;whatever functionality it is, it needs to be extended from a base service model, known as the *GridService* port type.

The *GridService* port type is like the base class (in the object-oriented languages) whose properties can be extended to suit different needs. This is the port type that must be implemented for any grid service in OGSI.

Available SDEs for the GridService port type

Various Service Data Elements (SDE) that are available in the *GridService* port type:

- Interface
- ServiceDataName (Service Data Element supported by this instance)
- factoryLocator (service locator to the factory that created the grid service instance)
- GridServiceHandle (GSH)
- GridServiceReference (GSR)

- findServiceDataExtensibility (element used for querying service data operations)
- setServiceDataExtensibility (element used for update service data operations)
- termination time (termination time for the service instance)

Various operations that can be performed using this interface are as follows.

FindServiceData operation

This queries the service data. FindServiceData operation takes an input query expression, which conforms to an inputElements declaration denoted by one of the FindServiceDataExtensibility SDE values. The service instance infers what to do based on the tag of the root element of this argument. Output to this operation is the result of the query.

The type of query expressions supported by an interface is expressed in instances FindServiceDataExtensibility SDE values. Therefore, a client can discover the query expression types supported by the instance by performing a FindServiceData request on the instance, using the queryByServiceDataNames with the name FindServiceDataExtensibility.

QueryByServiceDataNames operation

This operation queries the serviceDataValues of any particular ServiceDataElement contained in the ServiceDataName service in the service instance.

SetServiceData operation

If any ServiceDataElement is modifiable (`=True`) as per the service data declaration in the service instance, then you can use this operation to modify the values of the particular SDE. This operation takes an UpdateExpression and outputs the result of the update.

Similar to query expressions, the type of update expression supported by an interface is expressed in instance setServiceDataExtensibility SDE values. Therefore, a client can discover the query expression types supported by the instance by performing a FindServiceData request on the instance, using the queryByServiceDataNames element with the name of setServiceDataExtensibility.

SetServiceDataByNames operation

Use this operation to update any particular SDE values in Service Data Element of a service instance. The service data name is modifiable (`=true`) only when this operation succeeds. Note that this operation does not guarantee an update in any particular sequence. In case of failure, failed elements will be returned with a fault cause for each element failed. Also, depending on the mutability of the

element, the updating succeeds or fails. If the mutability value is `static` or `constant`, then setByServiceDataNames is not allowed. If the mutability value is `extendable`, then setByServiceDataNames must append to the new elements to the SDE's existing values.

DeleteByServiceDataNames operation

This operation deletes the Service Data Elements, which are listed in ServiceDataNames for the service instance. Also, the modifiable attribute should be `True` for the operation to successfully delete the element. Note that there is no guarantee that this operation will delete elements in a particular sequence. Deletion is not allowed for elements whose mutability value is `static`, `constant` or `extendable`. If the mutability value is `mutable`, then deleteByServiceDataNames will delete all elements with the SDE names requested.

RequestTerminationAfter/Before operations

Use the above two operations to change the termination time of the grid service instance to either the earliest time or a later time than the initial termination time set for the instance. The input to this operation is ExtendedDateTimeType Termination Time value and the return value is the Current Termination Time of type TerminationTime.

Destroy operation

As the name implies, this operation is used to destroy the grid service instance. Once the client uses this operation, the receiving end should either (a) initiate its own destruction and return the acknowledgement to the client service or (b) return failure indicating it was unable to destroy. After the initiation of destruction, the service instance should not respond to any further requests from client. The client, after receiving a successful destroy response, should not reply on the existence of service instance.

Factory port type

The factory grid service creates a new grid service instance on the client's request. The client uses a factory operation called createService to create a grid service and receives a locator, a GSH, in response. The client requests the minimum and maximum lifetimes for the service instance created. Depending on the governing local policies, this time may or may not be set to the created service instance. Along with GSH, the current termination time is also returned to the requesting service. The new grid service creation will fail if the time requested is not within the acceptable range. Apart from createService, there are no operations defined for the *Factory* port type.

The only Service Data Element defined for this port type is createServiceExtensibility. This includes a set of operation extensible declarations for the create service operation.

HandleResolver port type

HandleResolver is a port type which performs an operation, taking a GSH, and resolves the GSH using a GSH-to-GSR reference table; it returns a GSR in response. A grid service instance, which extends this *HandleResolver* port type capability, is called *handle resolver*.

There is one Service Data Element, called handleResolverScheme. This holds the set of GSH URIs that the handle resolver service instance can resolve.

There is one operation in the handle resolver, called findByHandle. This takes one or more GSH as input parameters (HandleSet) and resolves them. This operation returns a set of GSRs (Locator) corresponding to each GSH requested.

NotificationSource port type

The notification source is a grid service instance that implements the *NotificationSource* port type and can send any number of notification messages to any number of notification receivers called notification sink.

Notification message is an XML element sent from a notification source to sink.

Apart from the source, sink and message, there is a set of rules which governs when a notification will be sent from source to sink and what kind of message will be sent in what kind of situation (the SDE value changes within a service instance). These rules are called *subscription expressions*.

To initiate subscription, the client prepares and sends a subscription request that contains a subscription expression, locator of notification sink (where messages are to be sent), and an initial lifetime for the subscription. Once the serving end receives a subscription request, it creates a service instance called subscription, which implements the *NotificationSubscription* port type. Clients to manage the lifetime of the subscription and to discover properties of subscription use this port type.

There are two Service Data Elements in this port type implementation: NotifiableServiceDataName holds the set of Service Data Elements for the service instance for which subscription operation can be performed. SubscribeExtensibility is used to extend the capabilities of the subscription by defining new query expressions that are more powerful and more customized to a specific problem domain.

There are two operations that are allowed in this port type implementation. One is the Subscribe operation and the other is subscribeByServiceDataNames.

Subscribe operation

This operation is used by the notification sink to subscribe to specific service data change notifications from the *NotificationSource* port type. This operation needs three inputs: a subscription expression, a sink locator, and an initial expiration time for the subscription instance. This operation returns a locator to the subscription instance that was created to manage the subscription and current termination time for notificationSubscription instance.

SubscribeByServiceDataNames operation

This operation results in notification messages being sent whenever any named Service Data Elements change. There are two intervals when you subscribe to a notification source. One is maxInterval and the other is minInterval. MinInterval is the minimum time interval between the notification messages. MaxInterval is the maximum time interval between the notification messages; if there is no change to service data within the maximum time interval, then the last sent notification will be resent.

NotificationSubscription port type

As the notification sink subscribes to notification messages to a source, a notification subscription service instance will be created. This service instance implements the NotificationSubscription port type and extends the GridService port type. Clients use this subscription to manage the lifetime of the subscription and the properties of subscription.

This port type includes the following Service Data Elements.

SubscriptionExpression SDE

The current subscription expression managed by the subscription instance.

SinkLocator SDE

The locator of the Notification Sink to which messages are being delivered by this service instance.

There are no operations defined in this port type.

NotificationSink port type

The notification sink is a grid service that implements the *NotificationSink* port type and can receive any number of messages from any number of notification sources.

The *NotificationSink* port type does not have any Service Data Elements. It has one operation called deliverNotification, using which the subscription instance will be able to deliver messages to this notification sink service instance. This operation takes an XML element (message) containing the notification message. It does not return any value.

ServiceGroup port type

A service group is a grid service instance that maintains information about a group of other grid service instances. The grouping may be due to the nature of the service provided or may be an index of all services provided by an organization, or they may have no specific relationship.

The *ServiceGroup* port type provides an interface for representing a service group with zero or more member services. An Entry SDE identifies each service instance in the service group. The management capabilities (such as lifetime management, GSH management and other entry management functions) are handled by a separate service group port type called *ServiceGroupEntry*.

After the service group is destroyed, the client can make no assumptions about the existence of individual service group entries listed in that service group. Also, a grid service is not restricted to be only in one service group instance.

This port type establishes two Service Data Elements, as follows.

MembershipContentRule SDE

This SDE is an association between the port type (memberInterface) with a set of XSD element QNames (content). This SDE serves as a "data type invariant" for the Entry SDE values of the service group and for the content SDE value of the *ServiceGroupEntry* for the services in the *ServiceGroup*. This means that membership is restricted to only grid services that confirm these MembershipContentRule SDE values.

Entry SDE

There is one entry for every grid service in the service group. Each entry value is made up of three parts:

1. A locator that refers to the *ServiceGroupEntry* service instance that manages this entry
2. A locator to the member grid service instance referred by this entry
3. The content of the entry

There are no operations defined for this port type.

ServiceGroupRegistration port type

A grid service that extends the *ServiceGroup* port type and implements the *ServiceGroupRegistration* port type acts as an interface to manage the entries in the *ServiceGroup* port type.

There are addExtensibility and removeExtensibility SDE values available for this port type in order to manage add and remove operations supported by this port type.

There are two operations supported by the *ServiceGroupRegistration* port type: add and remove.

Add operation

The add operation adds a *ServiceGroupEntry* to the *ServiceGroup*. The input to this operation would be serviceLocator (pointing to the added member *GridService*), content (to associate with the service locator in the service group), TerminationTime (termination time for the added *ServiceGroupEntry* instance). The operation, upon success, returns a service locator and current termination time.

Remove operation

The remove operation removes a *ServiceGroupEntry* from the *ServiceGroup*. It takes an expression (MatchExpression) as input that matches the *ServiceGroupEntry* to be removed. This parameter is extensible. An acknowledgement will be returned after the successful completion of the operation.

ServiceGroupEntry port type

This port type is used to manage the properties and SDE values of the individual *ServiceGroupEntry*, which points to the actual grid service instances. This port type extends the *GridService* port type and implements the *serviceGroupEntry* port type.

There are two SDE for this port type. They are as follows.

MemberServiceLocator SDE

This SDE contains a service locator to the member grid service instance to which this entry pertains. The handles and references contained in this locator might change during the course of the service instance's lifetime.

Content SDE

This SDE holds information about the member service instance located using the MemberServiceLocator.

There are currently no operations defined for this port type.

4

Grid services development

This chapter describes a simplified method that embraces the complete development cycle of a grid service, providing straightforward guidelines on how to code, build and deploy a grid service in an arbitrary hosting environment.

A lengthy step-by-step approach with an illustrative example is taken so that no prior knowledge on developing grid services is required.

4.1 Introduction

The development of a grid service follows the concepts of Web services, but demands some extra requirements. Thus, a previous knowledge of Web services is welcome, but it is not enough, since specification and development are quite different. In any case, some background for Web services is presented in Appendix B, “Web service development” on page 213.

The Globus Toolkit 3.0 (GT3) offers the basic set of tools and libraries that allow the task of grid application development. This chapter is very much based on *GT3 documents and developments*.

Systems that are more complex may follow detailed developments procedures presented further in this document. This chapter has a simplified methodology that quickly leads to practical results.

In addition, although this sample is developed with grid tools, it is not a really grid example, since it does not have multiples clients, but a single client-server pair. A more realistic grid example, though simple, is presented in the next chapters.

4.1.1 Development machine

Here we describe the system we used for the development machine.

- CPU

  Building a grid service is a multi-step process but generally fairly quick to complete. Developers usually want high CPU speeds since faster is better. If developers will be running grid service tests directly on the development machine, please see the recommendation in the section below.

- Physical memory

  The memory used during compilation of a grid service code is defined by the **`javac`** overhead plus the number of Java Archives (JAR) files referenced by the module under compilation. Developer systems with 512 MB of RAM have comfortably served as GT3 development platforms, but as the grid service becomes larger, watch the system’s memory usage and upgrade when swapping is apparent. If developers will be running grid service tests directly on the development machine, see the recommendation in the section below.

Operating system and tools

GT3 service development can be performed on many operating systems including Linux, most Unix and recent Windows® versions. Services using only Java code should build on any certified Java platform.

- Required tools
  - Java 2 Standard Edition (J2SE) 1.3.1 Software Development Kit (SDK) certified for your platform.
  - Jakarta ant 1.5.
  - JAAS library as a separate download if you are using J2SE 1.3.
- Optional tools
  - Java 2 Standard Edition SDK 1.4+. This is required by some of the higher level services, like the Index Service and Execution Service, due to issues with some implementations of JDK 1.3.1.
  - Jakarta Tomcat 4.1.24 (4.0.6 has also been tested to work). A standalone Web service container is provided for testing purposes, but some users choose to deploy into Tomcat for production use.
  - Junit 3.8.1 if you want to run tests from the source.

4.1.2 Server machine

The server machine is part of the grid and it should provide the service. It must have GT3 installed on it.

- CPU

  The GT3 software itself is not computation-intensive, so the CPU of the GT3 server should be chosen with the computational requirements of the jobs it will run. If the CPU can adequately support the computational requirements of the deployed grid services, the Globus Toolkit software should not add any significant overhead.

- Physical memory

  The GT3 software itself is not memory-intensive, so any system with an amount of memory sufficient to support its deployed grid services will perform adequately.

Operating system and tools

GT3 runs on many operating systems including Linux, most Unix and recent Windows versions. The core of GT3 is pure Java and should run on any certified Java platform. Some additional components of GT3 use non-Java code and therefore are platform-specific. Because the list of supported systems is growing, please check the GT3 Web site.

- Bundled tools
- Third-party tools

  GT3 is shipped with a number of third-party tools. They are packaged with the GT3 distributions, so in general you don't have to worry about these. The

tools mentioned here are for reference purposes, it is not recommended that you replace any of these tools with another version you may already have.

  - Apache Axis post 1.1 final CVS checkout [06/18/2003]

    http://xml.apache.org/axis

  - Java CoG Kit post 1.1a CVS checkout [06/19/2003]

    http://www.globus.org/cog/java/1.1a

  - Apache Xerces 2.4.0 (JAXP 1.2)

    http://xml.apache.org/xerces2-j/

  - Apache-XML-Security-J 1.0.4

    http://xml.apache.org/security/index.html

- Required tools
  - Java 2 Standard Edition 1.3.1 Java Runtime Environment (JRE), or the SDK above if this is more convenient, certified for your platform.
  - Jakarta **ant** 1.5.
  - JAAS library as a separate download if you are using J2SE 1.3.
- Optional tools
  - Java 2 Standard Edition JRE 1.4+. This is required by some of the higher level services, like the Index Service and Execution Service, due to issues with some implementations of JRE 1.3.1.
  - Jakarta Tomcat 4.1.24 (4.0.6 has also been tested to work). A standalone Web service container is provided for testing purposes, but some users choose to deploy into Tomcat for production use.
  - A JDBC compliant database. The Reliable File Transfer (RFT) service and Replica Location Service (RLS) use a database back end. GT3 ships the Postgres JDBC driver, but other JDBC compliant databases should be usable. The database table initialization script uses Postgres-specific syntax and will require some porting effort to use with other databases.

4.1.3 Client machine

The grid service client machine makes use of the grid servers to get work done around the grid. The power of the machine should match the amount of work the client is expected to do: if it only runs a simple GT3 client application, almost any machine will work. As the complexity of the client applications grows, more CPU and RAM should be added until acceptable runtime performance is achieved. Note that much of what the client application will be doing is communicating with GT3 services on other machines, so the machine's network performance may in fact be more of a factor than CPU or RAM.

The client runs GT3-specific Java applications and these applications only work if all of the required GT3 and related JAR libraries are present on the client machine.

When you develop and run your GT3 client applications, you will quickly realize JAR files are missing because you will get `Class Not Found` error messages. It is possible, but tedious, to find out which JAR file the required class is in by examining each one with the Java **`jar`** command or a GUI tool that understands the JAR / ZIP file format.

This is a list of JAR files that were required for a very simple GT3 client application. Use this list as a starting point and add any additional JAR files which are required by your particular client application. The JAR files should be added to the CLASSPATH just before you execute your client, for instance:

```
CLASSPATH=.:ogsa.jar:axis.jar:jaxrpc.jar:commons-logging.jar:cog-axis.jar:
saaj.jar:commons-discovery.jar:cog-jglobus.jar:xmlParserAPIs.jar:
xercesImpl.jar:jgss.jar:xmlsec.jar:xalan.jar:log4j-core.jar:
jce-jdk13-119.jar
```

4.2 Grid development basic method

This section first introduces the reader to the major steps involved in creating a grid service, then goes through them with a simple but concrete example.

The grid service developed in this chapter has its functionality defined by a Java interface. This interface will be the starting point of the service's development cycle.

By means of a Java interface, one can specify a set of *methods*. Each method declaration corresponds to one specific operation and all the methods of an interface should provide a comprehensive description of its functionality.

Using tools for coding, building and deploying grid services

The use of development tools for automating and/or simplifying the process of coding, building and deploying a grid service is not only possible but also recommended in most cases.

For coding, there are several IDEs (Integrated Development Environments) such as Eclipse (or the commercialized IBM product WebSphere® Studio Application Developer (WSAD)), and JBuilder that offer a wide range of features dedicated to easing and speeding up the coding process. These tools have largely been employed by professional enterprises and have proven to increase the productivity of development teams by making their critical tasks less error-prone. At this point in time, these tools have not had additional functionality included to

transparently build grid services. We expect this to change quickly since the tools market is generally very quick to keep up with developer needs.

Despite their undeniable usefulness, these tools will not be addressed in this chapter since we are focusing on the development process itself.

Major steps

The major steps concerning grid service development are as follows:

Specifying	Phase to define the functionalities.
Coding	Phase to generate and adapt the code, WSDL and then the Java, using tools like Javac, Java2WSDL, DecorateWSDL and GSDL2Java.
Building	Phase to compile the code Java using Javac.
Packaging	Phase to aggregate compiled code into a package, using the JAR.
Deploying	Phase to place WSDD files in the right place and configure the container, using **`ant`**.
Testing	Phase to test the application, what can be done through some test code or some tool, like the Service Browser.

The next sections cover each of these phases.

4.2.1 Specifying

Before thinking about coding a grid service, a main concern should be the clear definition of the *functionality* that this grid service should expose (or export). This phase provides a low-level description of which tasks this service is capable of performing and which information is necessary for these tasks to be accomplished.

It is important to stress that knowing *what* a service is capable of doing is completely different from knowing *how* the service performs its tasks. There are virtually infinite ways of performing the same task and each of them has its own pros and cons. This separation is a cornerstone of object-oriented design and we fully exploit it in the grid service world.

4.2.2 Coding

This section addresses the main coding steps to be taken when developing a grid service. It discusses the basics of interface paradigm programming and introduces some simple rules of thumb that the developer has to be aware of to produce clean and working code.

The coding process has two distinct phases: produce or develop the WSDL code, then convert it to a programming language code.

Phase 1: Development of WSDL

When building a grid service, the developer must write a WSDL code file containing the description of the grid service's functionality.

If the developer does not already have the WSDL file, one possible way to begin is to first describe Java interfaces, and then convert them to WSDL. If this is the approach adopted, after defining the service interface in Java, it is necessary to compile it to generate the service stubs, and then transform this code into WSDL.

For the compilation process, we have two options: to compile all the classes from the directory where the root package com/ is placed or to modify the environment variable CLASSPATH so that we can compile the classes from anywhere.

Then, the developer can use tools to automatically generate the WSDL file from a Java interface, like Java2WSDL, detailed in Appendix C, "Java2WSDL and WSDL2Java" on page 231 and discussed in Appendix B, "Web service development" on page 213.

However, since the grid has specific requirements, and these are not incorporated in the specification WSDL V1.0, one more step is necessary to perform some modifications on the WSDL file, resulting in the Grid WSDL (GWSDL) file. In the future, if the WSDL 1.2 specification actually includes all the grid requirements, this step may be not necessary.

These conversion procedures ease the development process because they free the developer from dealing with WSDL files, but they have their drawbacks: as the degree of abstraction increases when automatically generated content is incorporated to a process, the degree of control that the developer has over the process tends to decrease.

A common approach is to use this automatically generated WSDL file as a starting point for describing a grid service. As new features are incorporated, the service becomes more sophisticated and the WSDL file is manually modified to represent these changes.

Phase 2: Achievement of the Java code: the stubs

The second phase of coding consists of transforming the WSDL (or the GWSDL) into a specific programming language code, in this case, Java, implying stubs generation.

In the same way as for the Web services development process, after obtaining a WSDL description of the grid service, you can generate the *stub files*. These files

are also generated automatically and provide the *glue* between the service itself and its clients, so that invocations to the service appear to the clients as local method calls.

The client side stub of a service acts as a proxy for the service and hides the communication protocol and data conversions required to transmit the messages from the client to the server. The client proxy is compiled and linked with the client side code and provides a Java programming interface to the client. The grid service messages are transferred as SOAP messages as specified in earlier chapters. However, the client side proxy provides a Java programming interface to the client and shields the client from the details.

Similarly, the server-side stub hides the data conversion and communication protocol specifics from the server implementation. The server-side stub invokes the server implementation by passing the incoming data values in the data types of the Java language and converts the output data values of the server implementation from the Java data types to the communication format before transmitting them to the client.

When developing Java grid services for GT3, a set of tools can be used to generate and manage the stub files from the GWSDL file, including the GSDL2Java. Note that this is not the same as used for Web services (the WSDL2Java, seen in Appendix B, “Web service development” on page 213) because the WSDL file was converted to GWSDL. This process takes as input the GWSDL file and produces the Java and WSDD files for both client and server sides.

4.2.3 Building

When building a grid service, a series of compilation steps have to be taken in a very specific way. Although some compiling has already been done in the coding phase, it was only performed in order to prepare the final code. It is in the building phase that the final code, in this case Java, is actually compiled. This is also called service implementation.

Service implementation

The grid service implementation itself does not require any specific procedure other than those required by Web services. In the case of a Java grid service, all that should be done is to provide a grid-enabled implementation for all the methods that define the service's functionality.

When developing Java grid services for the GT3 container, there are two options for implementing such functionality: delegation and inheritance. Both approaches are discussed in detail in the GT3 documentation (see

http://www-unix.globus.org/toolkit/documentation.html) but, in simple terms, the most natural and flexible way is delegation.

In order to delegate to a Java class the ability to run in the GT3 container, this class must implement the org.globus.ogsa.OperationProvider class. In this way, all the methods required by the GT3 container to properly manage the class are embedded in it, making it grid-enabled. One important detail about delegation is that you can spread the implementation of the service's methods across multiple classes, which might be very useful when dealing with legacy code.

The inheritance approach is simpler in the sense that it requires only the implementation of the service's interface for the service class to become grid-enabled. The remaining methods, required by the grid container, are inherited from a standard class (namely, the org.globus.ogsa.impl.ogsi.GridServiceImpl class) which has implemented the OperationProvider interface itself. The main drawback of this approach is that, since the Java language does not support multiple inheritance, it will not be possible for the service class to inherit from any other class, which might be too restrictive in many cases. More details and alternatives for this problem can be found in Appendix E, “Delegation” on page 243.

Important: All the Java files must be placed in directories that can be reached by the **javac** compiler. This can be accomplished either by inserting the root source directory into the CLASSPATH environment variable or by running the compiler from this directory.

The steps for compiling grid services written in other programming languages may be similar, but will not be addressed in this document.

4.2.4 Packaging

Before deploying a grid service into its container, all its Java classes and supplementary files must be placed inside a Grid Archive (GAR) file. A GAR file is built with **jar** or **zip** tools, but has a very specific file layout, and a .gar extension.

The rules that the developer has to be aware of when building a GAR file are the following:

1. The Web service deployment descriptors (WSDD) file must have its name set to `server-deploy.wsdd` and must be placed in the archive’s root directory.
2. The WSDL/GWSDL file must be placed inside a directory named *schema* which lives in the archive’s root. It is also possible to place this file inside a

sub-directory of the schema directory, as long as its location is correctly specified in the WSDD file.

3. There is no specific rule for placing the remaining files but it is a good organizational principle to keep the implementation files separate from the automatically generated stubs so that they can be easily redistributed to the clients. Therefore, one option is:
   a. Place all the stubs in a JAR file and place this file in the archive's root.
   b. Place all the implementing files in a separate JAR file and place this file in the archive's root as well.

Tip: The name of the JAR files where the stubs and implementation files are placed does not matter since they will be automatically scanned and loaded into memory by the grid container. However, you are advised to give them meaningful names, such as <service-name>-stubs.jar for the stubs JAR file and <service-name>-impl.jar for the implementation JAR file.

4.2.5 Deploying and undeploying

GT3 Web services use the same WSDD files as are used for Web services deployment. The structure of a WSDD file for GT3 remains unchanged from the Axis version.

The process for deploying and/or undeploying a grid service in a grid container may vary from container to container. This process consists of copying the service's files into locations where the container can find them when they are required and from where they can be removed when the service is undeployed. It also processes the WSDD files so that the container's configurations are set up to include or exclude this service into/from the set of all active services for that container.

Deploying a service into GT3 is very straightforward once it can be done by the **ant** tool from the GT3 root directory.

```
ant deploy -Dgar.name=<fully qualified GAR file pathname>
```

Similarly, to undeploy a grid service from the GT3 container, run the following command from the GT3 root directory:

```
ant undeploy -Dgar.id=<service name>
```

4.2.6 Testing

After deploying a grid service into a container, there are basically two ways to test it: using the Service Browser or developing a client code.

The simplest and quickest is by means of the **`Service Browser`** tool, which is discussed in Appendix F, “Service Browser” on page 253.

The second, more laborious, but most comprehensive way, is to implement a simple client that tests each of its exported methods in a more specific fashion. In this case, such a client should get a reference to a service factory, create its own instance and issue calls to each of its methods in a convenient way.

The sample code shown in Figure 4-1 on page 48 can be used as a starting point for creating such clients. It includes all the necessary operations for creating a service instance and issuing method calls to this instance. In order to run the client, all the required libraries (JAR files) must be included in the CLASSPATH environment variable.

Note that this client is meant for a non-secure grid server, which means that no certificates are required.

```
package <my client package>;

import <my service's stubs package>.<my service's port-type class>;
import <my service's stubs package>.<my service's grid locator class>;
import org.globus.ogsa.utils.GridServiceFactory;
import org.gridforum.ogsi.Factory;
import org.gridforum.ogsi.LocatorType;
import org.gridforum.ogsi.GridService;
import org.gridforum.ogsi.OGSIServiceGridLocator;
import java.net.URL;

public class SampleClient {
   public static void main(String[] args) {
      try {
         // Set the service address
         String addr = "http://<grid server>/<service location>";
         URL GSH = new java.net.URL(addr);

         // Get a reference to the service Factory
         OGSIServiceGridLocator gl = new OGSIServiceGridLocator();
         Factory fp = gl.getFactoryPort(GSH);
         GridServiceFactory gsf = new GridServiceFactory(fp);

         // Create a new instance of the service
         LocatorType lt = gsf.createService();

         // Get a reference to the service port-type.
         <service name>GridLocator sgl= new <service name>Locator();
         <service name>PortType spt= sgl.get<service name>(lt);

         // Invoke remote methods
         spt.<method call>

         // Destroy the service instance
         GridService gs = gl.getGridServicePort(lt);
         gs.destroy();
      } catch(Exception e) {
         System.err.println("There was an error executing the client");
         e.printStackTrace();
      }
   }
}
```

Figure 4-1 Sample grid client code skeleton used for testing

4.3 Grid services development sample

Following previous concepts and development steps, this section presents a simple but complete grid service.

The sample grid service to be developed in this chapter is a distributed client-server application written in Java. The server returns the “message of the day” to any client's invocation.

This is a direct extension of the Web service developed in Appendix B, “Web service development” on page 213, but in addition, it keeps track of the messages already sent to each of the clients, providing a new message for every method call, regardless of the current date.

Although the applications are so similar, and that grid has taken much in the way of Web services knowledge and some tools, be careful, because grid development requires some different procedures.

The development follows the methodology and the language is Java.

4.3.1 Essentials

The simple grid service may be developed with the use of the following elements:

1. A Java development kit, with a Java compiler and virtual machine.
2. The JAR files located inside the lib/ directory of any GT3 installation.
3. The JAR pathnames included in he CLASSPATH environment variable.
4. The contents of the schema/ directory of any GT3 installation.

As you can see, a GT3 installation is *not* required, only its libraries. A copy of all the required files should be available so that the tools for generating WSDLs and stubs can be accessed.

4.3.2 Specifying: defining the service’s functionality

The service’s functionality for this sample is presented in Figure 4-2 on page 50.

```
package com.ibm.itso.grid.gt3.motd.common;

/**
 * This interface defines the sample service functionality.
 * It defines a single method from where callers can get text messages
 */
public interface MOTDSI
{
    public String getMOTD();
}
```

Figure 4-2 Message of the day service interface

As in every Java project, the Java file where this interface is declared must be named MOTDSI.java and must be placed in a directory structure that reproduces the package structure declared in the very beginning of the file. So, you are expected to create a directory com/ that houses the sub-directory ibm/ that houses the sub-directory itso/ and so on until the sub-directory common/, where the file is finally placed.

4.3.3 Coding sample

This sample considers that the first specification is done in Java. Thus, it is necessary to convert the Java code into WSDL code and into GWSDL code.

The WSDL for grid services

In order to compile the Java service interface, we chose to compile all the classes from the package /com directory, for the sake of simplicity since we will not have to deal with environment variables configuration issues, which may vary from platform to platform.

Thus, from the directory that houses the com/ sub-directory, issue:

```
javac com/ibm/itso/grid/gt3/motd/common/MOTDSI.java
```

This will generate the compiled MOTDSI.class file inside the same directory where MOTDSI.java is.

At this point, a copy of the directory <OGSA location>/schema/, along with all its contents, should be made in the same directory where the sub-directory com/ lives. Additionally, a sub-directory motd/ should be manually created inside this schema/ directory.

Thereafter, the WSDL file can be generated by issuing:

```
java org.apache.axis.wsdl.Java2WSDL -S MOTD -P MOTDPortType -o
schema/motd/MOTD.wsdl -l http://localhost/ogsa/services/MOTD -y WRAPPED -u
LITERAL -n http://common.motd.gt3.grid.itso.ibm.com/stubs
com.ibm.itso.grid.gt3.motd.common.MOTDSI
```

This command is basically the same as was issued for the sample Web service. Further information about each of its command line parameters can be found in Appendix B, “Web service development” on page 213.

Important: If you are issuing this command in a machine where the GT3 is installed, then you will not have to deal with CLASSPATH configuration since it may already have been properly set for running GT3. However, if you are issuing the command in a non-GT3 machine, you will have to copy all the required JAR files to your local system and place them in your CLASSPATH environment variable. The required JAR files are in the <OGSA location>/lib directory.

After that, move to the directory schema/motd/, where the WSDL file has just been generated, and issue:

```
java org.globus.ogsa.tools.wsdl.DecorateWSDL ../../ogsi/ogsi_bindings.wsdl
MOTD.wsdl
```

This command, which is provided as an OGSA tool by the Globus Alliance, transforms the WSDL file into a decorated WSDL file, actually the GWSDL file, with all the additional information required by grid services. Remember that this is necessary in the WSDL 1.1 specification and may not be necessary anymore in the new version WSDL 1.2 if it includes all grid requests.

The Java code: stubs for grid services

Having the decorated WSDL file, we are able to generate all the service stubs. This is done by issuing:

```
java org.globus.ogsa.tools.wsdl.GSDL2Java schema/motd/MOTD.wsdl
```

This command creates the new directory structure stubs/ in the common/ directory and places all the stub files inside it.

4.3.4 Building the sample: service implementation

Once the stubs are generated, it is possible to write both service and client implementations. Figure 4-3 shows the code that implements our sample grid service.

```
package com.ibm.itso.grid.gt3.motd.server;
import com.ibm.itso.grid.gt3.motd.common.stubs.MOTDPortType;
import org.globus.ogsa.impl.ogsi.GridServiceImpl;
import java.rmi.RemoteException;

/**
 * This class implements a grid service that provides text
 * messages to its clients from a pre-defined list of messages.
 * Its main difference from a web-service is that it.s statefull,
 * as it keeps track of the last message that has
 * been delivered to each of the clients.
 */
public class SimpleServer extends GridServiceImpl implements MOTDPortType {
   private int nextMessage = 0;
   String motds[] = {<fill in message strings here>};

   public SimpleServer() {
      super("MOTD Constructor");
   }

   public String getMOTD() throws RemoteException {
      String motd = motds[nextMessage];
      nextMessage++;
      nextMessage = (nextMessage) % 10;
      return motd;
   }
}
```

Figure 4-3 Grid service sample code

As can be seen, this code looks fairly similar to the Web service implementation code without the grid presented in Appendix B, “Web service development” on page 213. The main difference is that each instance of this grid service will have its own state. Here, the state is simply represented by the local attribute `nextMessage`, which points to the index of the next message to be delivered.

The service implementation code can be compiled by issuing the following command from the directory where the com/ sub-directory has been placed.

```
javac com/ibm/itso/grid/gt3/motd/server/SimpleServer.java
```

Then the following command compiles all the stub classes:

```
javac com/ibm/itso/grid/gt3/motd/common/stubs/*.java
```

4.3.5 Packaging the sample

Once all the service classes have been compiled, they should be packaged.

To group all the stub files in a single JAR package, issue:

```
jar cvf motd_stubs.jar com/ibm/itso/grid/gt3/motd/common/stubs/*.class
```

Similarly, to create a JAR file containing the service implementation class, issue:

```
jar cvf motd_impl.jar com/ibm/itso/grid/gt3/motd/server/*.class
```

Now we have everything we need to create the GAR file. This can be accomplished by issuing the following command:

```
jar cvf motd.gar motd_impl.jar motd_stubs.jar server-deploy.wsdd
server-undeploy.wsdd schema/
```

This command creates the motd.gar archive with the required JAR files, the service-deploy.wsdl and service-undeploy.wsdd and the MOTD.wsdl file along with its directory path.

4.3.6 Deploying the sample

In this section, the service will be deployed. The undeploy method will also be shown.

The WSDD Deployment Descriptor file

Now we can analyze server-deploy.wsdd and server-undeploy.wsdd files, generated by the **GSDL2Java** command and already packaged. For our sample grid service, their contents are rather standard as you can see in Figure 4-4 on page 54 and Figure 4-5 on page 54, respectively.

```
<?xml version="1.0"?>
<deployment name="defaultServerConfig"
xmlns="http://xml.apache.org/axis/wsdd/"
xmlns:java="http://xml.apache.org/axis/wsdd/providers/java">

<service name="MOTD/MOTDFactory" provider="Handler" style="wrapped">
   <parameter name="name" value="MOTD Factory"/>
   <parameter name="instance-name" value="MOTD Instance"/>
   <parameter name="instance-schemaPath" value="schema/motd/MOTD.wsdl"/>
   <parameter name="instance-ClassName"
      value="com.ibm.itso.grid.gt3.motd.MOTD.MOTDPortType"/>
   <parameter name="instance-baseClassName"
      value="com.ibm.itso.grid.gt3.motd.server.SimpleServer"/>

   <!-- Start common parameters -->
   <parameter name="allowedMethods" value="*"/>
   <parameter name="persistent" value="true"/>
   <parameter name="className" value="org.gridforum.ogsi.Factory"/>
   <parameter name="baseClassName"
      value="org.globus.ogsa.impl.ogsi.PersistentGridServiceImpl"/>
   <parameter name="schemaPath"
      value="schema/ogsi/ogsi_factory_service.wsdl"/>
   <parameter name="handlerClass"
      value="org.globus.ogsa.handlers.RPCURIProvider"/>
   <parameter name="factoryCallback"
       value="org.globus.ogsa.impl.ogsi.DynamicFactoryCallbackImpl"/>
   <parameter name="operationProviders"
      value="org.globus.ogsa.impl.ogsi.FactoryProvider"/>
</service>

</deployment>
```

Figure 4-4 Grid service deployment descriptor

```
<?xml version="1.0" encoding="UTF-8" ?>
<undeployment name="defaultServerConfig”
   xmlns="http://xml.apache.org/axis/wsdd/"
   xmlns:java="http://xml.apache.org/axis/wsdd/providers/java">
   <service name="MOTD/MOTDFactory" provider="Handler" style="wrapped" />
</undeployment>
```

Figure 4-5 Grid service undeployment descriptor

Deploying the service

Finally, the service can be deployed to the GT3 container by issuing the following command from the GT3 root directory:

```
ant deploy -Dgar.name=<fully qualified GAR path>/motd.gar
```

This command places every file in its proper directory.

4.3.7 Testing sample

For this service to be tested, we developed a simple client that creates a service instance and performs a number of calls to the service exported method, as discussed in 4.2.6, “Testing” on page 47.

The client code can be derived from the standard code presented in Figure 4-1 on page 48, resulting in the code shown in Figure 4-6 on page 56.

```
package com.ibm.itso.grid.gt3.motd.client;

import com.ibm.itso.grid.gt3.motd.MOTD.MOTDPortType;
import com.ibm.itso.grid.gt3.motd.MOTD.MOTDServiceGridLocator;
import org.globus.ogsa.utils.GridServiceFactory;
import org.gridforum.ogsi.Factory;
import org.gridforum.ogsi.LocatorType;
import org.gridforum.ogsi.GridService;
import org.gridforum.ogsi.OGSIServiceGridLocator;
import java.io.DataInputStream;
import java.net.URL;

public class SimpleClient {
   public static void main(String[] args) {
      try {
         // Read the service address in first argument
         URL GSH = new java.net.URL(args[0]);

         // Get a reference to the service Factory
         OGSIServiceGridLocator gl = new OGSIServiceGridLocator();
         Factory fp = gl.getFactoryPort(GSH);
         GridServiceFactory gsf = new GridServiceFactory(fp);

         // Create a new instance of the service
         LocatorType lt = gsf.createService();

         // Get a reference to the service port-type.
         MOTDServiceGridLocator sgl = new MOTDServiceGridLocator();
         MOTDPortType spt = sgl.getMOTDService(lt);

         for (int i=0; i<15; i++) {
            // Invoke remote method
            System.out.println(spt.getMOTD());
            System.in.read();
         };

         // Destroy the service instance
         GridService gs = gl.getGridServicePort(lt);
         gs.destroy();
      } catch(Exception e) {
         System.err.println("There was an error executing the client");
         e.printStackTrace();
      }
   }
}
```

Figure 4-6 Grid client code

After creating its service instance, this client keeps calling the service's method as long as the **Enter** key is pressed between two iterations. This allows for the verification of the stateful nature of the service instances, since multiple clients running at the same time may call the service's method in any arbitrary order.

This code may be compiled by issuing the following command:

```
javac com/ibm/itso/grid/gt3/motd/client/SimpleClient.java
```

It may be started by issuing the following command:

```
java com.ibm.itso.grid.gt3.motd.client.SimpleClient
http://<servername>:8080/ogsa/services/MOTD/MOTDFactory
```

Important: Before executing the client, the grid service container where our service was deployed must be running. In GT3, this can be accomplished by issuing the **globus-start-container** command from the <OGSA location> directory.

If several clients are started simultaneously, it is possible to verify that each one has its own grid service *context*, which means that the execution of a client does not interfere with the order according to which messages are delivered to another client.

In spite of the simplicity of this example, the steps followed for the development of our sample grid service may remain unchanged regardless of the complexity of the grid service being developed.

5

Major features of grid services

The preceding chapters introduced entry-level concepts and a basic approach to developing grid services. The focus of this chapter is to incrementally build on that base knowledge by gradually introducing the topics that demonstrate the OGSI and GT3 features of grid services and show how these grid service features address the shortcomings of the Web services model.

5.1 Introduction

Grid applications have some special features, which were mentioned in Chapter 3, “Open Grid Services Architecture” on page 21. This chapter will detail the most important features of grid programming: Factory, Service Data Elements, Life cycle and Notification.

5.2 Factory

Grid services implement a factory approach which is similar to the factory concept in object-oriented design and object-oriented programming in Java. A factory is a persistent service which creates instances with which clients can interact. In object-oriented programming terms, a factory is used to create instances of a class. The factory is also used to isolate the creation of objects of a particular class into a single place so that new features or functions can be added without widespread code changes.

In a grid context, a factory creates service instances and has a registry to keep track of those instances and to enable service discovery by clients or other services.

Clients typically first locate the factory, and then request the creation of a service instance. On request, a factory creates an instance of a grid service and returns a Grid Service Handle (GSH) and a Grid Service Reference (GSR) to the client.

The GSH is a unique identifier and the client uses it to communicate with the service instance. No further communication from the client necessitates the factory and communication is established directly with the service instance.

The service instance maintains state data relative to the client which invoked it. Typically, clients will have their own grid service instance with which to interact but it is possible for multiple clients to interact with the same grid service instance. The instance is destroyed when the client(s) no longer has a need for it. Optionally, a termination time can be specified. When the grid service has been idle for the length of time specified by the parameter *termination time*, it is destroyed. When it is destroyed, it frees any resources it obtained during its lifetime.

A typical client scenario includes the following steps:

1. Client discovers a factory by querying the registry service
2. Client calls a factory operation to create an instance of a grid service
3. Factory creates a new instance of the grid service
4. Factory returns the GSH of the new grid instance to the client

5. Client and service interact as result of the initial call

The example in Figure 5-1 illustrates the concept of a client requesting an instance from the service factory. The service factory creates an instance and returns the GSH or URL of the service instance to the client. The client uses this GSH to communicate directly with the service instance.

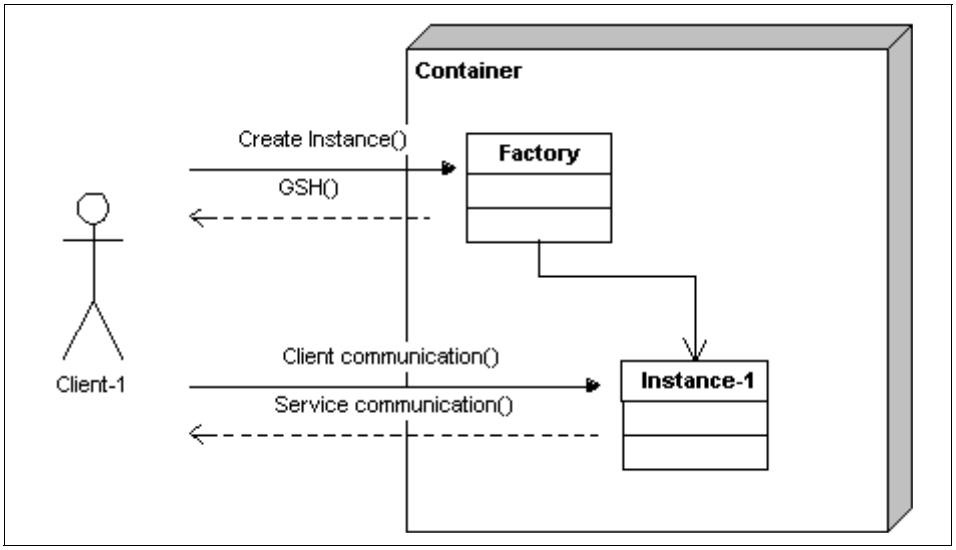


Figure 5-1 Client requests an instance from the factory

5.3 Service Data Elements

As noted in the preceding sections and with the previous examples, an important differentiation between grid services and Web services is the ability of a grid service to maintain state information. Another important differentiation is that grid services are transient. Whereas a Web service is persistent and all clients interact with that Web service, a grid service factory creates an instance for each client to interact with. The persistent state data associated with that grid service instance is relative to the client that invoked the service. This allows for stateful interactions between the client and the service.

The service data is a structured collection of information that is associated with an instance of a grid service to expose a grid service instance's state data to service requestors. The service data must be easy to query, so that grid services can be classified and indexed according to their service characteristics.

Each instance of a grid service has an associated service data set which contains Service Data Elements (SDE); these can be of different types. However, Service Data Elements of the same type always contain the same type of information.

Every grid service instance has standard SDEs by default. Consider the scenario where a service data set has two service data types. One service data type (type A) is composed of service data that describes resource information such as the architecture, speed, operating system level and available disk space information. The second service data type (type B) contains service data that describes service in terms of quality of service and degree of precision. All service data sets which contain service data type B will contain the same data elements (quality of service and degree of precision), however, the value associated with those Service Data Elements will be unique to the particular grid service instance. All service data sets which contain service data type A will contain the Service Data Elements associated with that resource (architecture, speed, operating system level and available disk space information).

Each Service Data Element must have a locally unique name and can contain name-value pairs, concrete Java types, user-defined types, or any combination thereof.

SDEs can be static or dynamic. Static service data is defined as part of the service interface definition. Dynamic service data is added to the service instance. To use dynamic service data, clients must be able to get a list of Service Data Elements at runtime. This is accomplished by using the findServiceData method defined in the grid service interface. Example 5-1 returns all Service Data Elements for an individual service instance. Notice that the only parameter is the GSH (or the URL of the service instance).

Example 5-1 findServiceData

```
findServiceData
http://192.168.0.102:12080/ogsa/services/base/index/IndexFactoryService
produces the following output:
<ns1:serviceDataSet xsi:type="ns1:serviceDataSet">
<ns1:serviceData xsi:type="ns1:serviceData"
ns1:availableUntil="2002-12-15T16:13:01.701Z"
ns1:goodFrom="2002-12-14T16:13:01.701Z"
ns1:goodUntil="2002-12-15T16:13:01.701Z"
ns1:name="ns2:NotifiableServiceDataNames"
xmlns:ns2="http://ogsa.gridforum.org/notification/notification_source/
definitions"/>
.
.
.
```

```
<ns1:serviceData xsi:type="ns1:serviceData"
ns1:availableUntil="2002-12-15T16:13:01.700Z"
ns1:goodFrom="2002-12-14T16:13:01.700Z"
ns1:goodUntil="2002-12-15T16:13:01.700Z" ns1:name="ns3:GridServiceHandles"
xmlns:ns2="http://ogsa.gridforum.org/notification/notification_source/
definitions"
xmlns:ns3="http://ogsa.gridforum.org/service/grid_service/definitions"
xmlns:ns4="http://ogsa.gridforum.org/factory/factory/definitions"
xmlns:ns6="http://ogsa.gridforum.org/registry/registration/definitions"
>
<ns1:serviceHandle
xsi:type="ns1:HandleType">http://128.9.72.46:8080/ogsa/services/base/
index/IndexFactoryService</ns1:serviceHandle>
</ns1:serviceData>
</ns1:serviceDataSet>
```

Example 5-1 on page 62 returns all Service Data Elements for an individual service instance. In order to return a specific Service Data Element for a particular service instance, the **findServiceData** command must include GSH (or URL of the service instance) along with the Service Data Element name within the instance that is being requested. See Example 5-2 for details on returning a specific SDE from the service instance.

Example 5-2 findServiceData query to locate the gridServiceHandle

```
findServiceData
http://128.9.72.46:8080/ogsa/services/base/index/IndexFactoryService
GridServiceHandles
produces the following output:
<ns1:serviceData xsi:type="ns1:serviceData"
ns1:availableUntil="2002-12-15T16:13:01.700Z"
ns1:goodFrom="2002-12-14T16:13:01.700Z"
ns1:goodUntil="2002-12-15T16:13:01.700Z" ns1:name="ns2:GridServiceHandles"
xmlns:ns2="http://ogsa.gridforum.org/service/grid_service/definitions">
<ns1:serviceHandle
xsi:type="ns1:HandleType">http://128.9.72.46:8080/ogsa/services/base/
index/IndexFactoryService</ns1:serviceHandle>
</ns1:serviceData>
```

A developer can include SDEs in a grid service. In addition to user-defined service data, each grid service has a set of common service data. The service data depends on the grid service port type which is implemented. The common service data is populated by the Globus core framework. The common service data describes characteristics of the grid service such as the GSH of the instance. As explained in 3.3, “Open Grid Services Infrastructure (OGSI)” on page 28, the following Service Data Elements are part of the *GridService* port type:

- gridServiceHandle
- factoryLocator
- terminationTime
- serviceDataName
- interfaces
- gridServiceReference
- findServiceDataExtensibility
- setServiceDataExtensibility

Java developers wishing to implement service data might think that the SDEs need to be added to the grid service interface. It is recommended to have a separate Java class for each service data type. In this case, each Service Data Element is an instance of one of these Java classes. Each attribute associated with service data should have the appropriate access methods. These are typically called getters and setters. This portion of code can be generated from a service data description which is specified in an XML schema document, which is imported into the GWSDL description of the grid service. Example 5-3 demonstrates this.

Example 5-3 GWSDL document with service data description

```
...
<complexType name="MyDataType">
   <sequence>
      <element name="myValue" type="int"/>
      <element name="myOperation" type="string"/>
   </sequence>
</complexType>
...
```

5.4 Life cycle

Most entities have a life cycle. This typically refers to the states between the object's creation and destruction. Life cycle management is very important, especially in a robust environment where services should be capable of resuming operations in the event of a server or container restart. In order to be capable of resuming operations after a container restart, services must support checkpoints and persisting of its state information. The ability to retrieve the previous state and continue execution or processing from that state forward must also be supported. In other words, the service should be able to continue operations after the container has been restarted with the same state and values that it had when the container was stopped. To support this, the service must manage itself during critical points in its life cycle. A grid service should save the state of any internal values prior to being destroyed and reload the previously

saved state during creation. The Globus Toolkit and the IBM Grid Toolbox provide tools to aid in the life cycle management of grid services. These tools include what are known as callback methods. These callback methods are called or invoked during critical points of a grid services life cycle.

These critical points during the grid service life cycle include:

- preCreate - called when a grid service starts the creation process, prior to loading configuration data
- postCreate - called when a grid service has been created and loaded its configuration data
- activate - called when a grid service is activated or loaded into memory space
- deactivate - called before a grid service is deactivated or paged out of memory
- predestroy - called before a grid service is destroyed

Callbacks methods can be implemented in a grid service by implementing the GridServiceCallback interface. The following code segment (Example 5-4) illustrates implementing callback methods.

Example 5-4 Callback method

```
import org.globus.ogsa.GridServiceCallback;

public class myClass implements GridServiceCallback {

   public void preDestroy(GridContext context) throws GridServiceException {

      logger.info("Grid Service destroy method invoked, saving state data…");
      // save the internal state of the service here

   }
```

As with most programming concepts, there are numerous ways to implement this technology. Similar to the delegation approach in the preceding section, the container can handle some of this functionality by strategically placing statements in the deployment descriptor to enable the life cycle monitor. The life cycle monitor is a class which implements the ServiceLifecycleMonitor interface. The ServiceLifecycleMonitor interface contains callback methods which are called at specific points in a grid service's life cycle. The following example illustrates concepts that were previously presented during our discussion of persisting a service so that it can resume operations after a container restart. As mentioned previously, to support this requirement, the service must log its internal state prior to being destroyed. The service must reload the previously saved state during object creation and be capable of resuming operations from

that point. See Example 5-5 for details on implementing the create and preDestroy callback methods.

Example 5-5 Implementing callback methods

```
Package com.ibm.itso.gt3.lifecycle.impl;

import org.globus.ogsa.GridServiceException;
import org.globus.ogsa.ServiceLifeCycleMonitor;
import org.globus.ogsa.GridContext;
import org.apache.commons.logging.log;
import org.apache.commons.logging.logFactory;

public class myClass implements ServiceLifecycleMonitor {

   static Log logger = LogFactory.getLog(myClass.class.getName());

   public create(GridContext context) throws GridServiceException {

      logger.info("myClass instance is being created");
      // check for saved state and retrieve if appropriate
   }

   public preDestroy(GridContext context) throws GridServiceException {
      logger.info("myClass instance is being destroyed");
      // saved the state of this instance
   }
```

Remember that to view the log output, the following files must be updated accordingly:

- ogsiLogging.properties
- ogsilogging_parm.properties

If the grid service does not implement the GridServiceCallback interface, then the other approach to implementing callback methods is to update the deployment descriptor so that the container calls certain life cycle monitors when specific events take place. Life cycle parameters can be specified in the deployment descriptor. For example, the instance-deactivation parameter allows the developer to specify the amount of time in milliseconds that a particular instance can be idle prior to deactivation. In the following example, the instance will be deactivated after it has been idle for 15 seconds. This property is important since services are deactivated by default, but once activated, they are active indefinitely. Developers or application architects should consider how long a service instance should remain active if it is not being used. If this is not considered, additional system resources are being consumed. Deactivation will free system resources and when the instance is invoked again, it will become

active. Example 5-6 illustrates instance deactivation declared in the deployment descriptor.

Example 5-6 Instance deactivation

```
<parameter name="instance-deactivation" value="15000"/>
```

5.5 Notifications

Notifications are a useful mechanism for tracking changes to service data. A party interested in a particular Service Data Element registers to be notified if that value changes. The interested party, to which the notification is sent, is called the notification sink. The service containing the Service Data Element of interest and which generates the notification to interested parties or subscribers is called the notification source.

Figure 5-2 on page 68 illustrates the notification subscription flow of events. An application or grid service is interested in a particular Service Data Element of another grid service (notification source). The interested party (notification sink) generates a notification subscription which includes a subscription expression that describes which Service Data Element(s) and associated changes are of interest. The notification subscription is sent from the interested party (notification sink) to the grid service that contains the Service Data Element of interest. In other words, the notification sink calls the subscribe operation on the notification source. Upon receipt of the notification subscription, the grid service (notification source) creates a subscription service to manage the subscription and its associated information. The notification source returns the handle of the subscription service to the requester (notification sink). The handle is used by the notification sink to manage the subscription lifetime. Notification messages are then sent to the notification sink when the Service Data Element, specified in the notification subscription, changes. Notifications continue for the lifetime of the notification subscription. The notification sink and the subscription service can interact to perform lifetime management tasks.

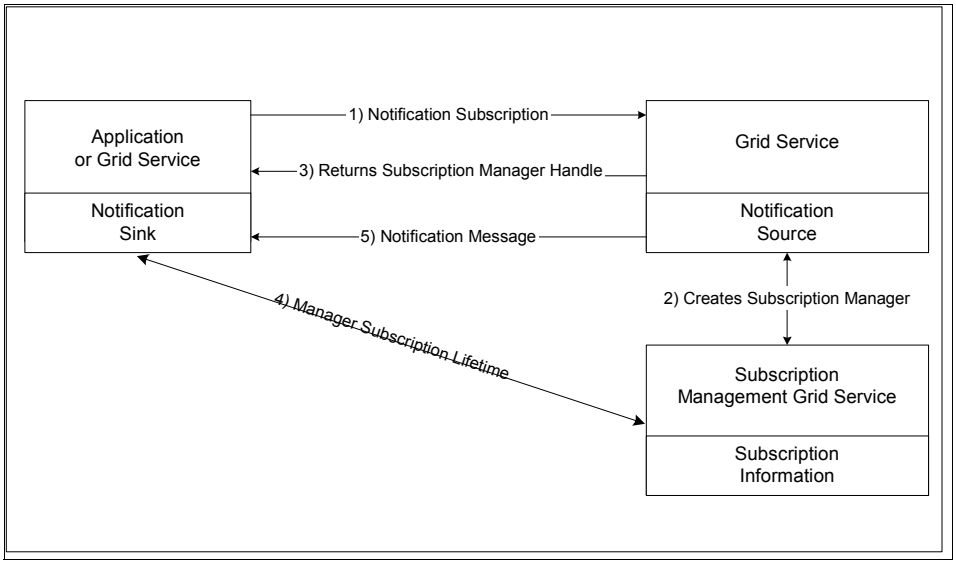


Figure 5-2 Notification subscription flow

1. An interested service subscribes to be notified if a Service Data Element changes. The interested service that will be notified is called the notification sink. The grid service that received the notification subscription is called the notification source.
2. The notification source creates a subscription manager instance.
3. The notification source returns the handle of the subscription manager instance to the notification sink.
4. The notification sink can use the subscription manager handle to manage the subscription lifetime.
5. When the condition specified in the notification subscription is met, a notification message is sent to the notification sink.

The notification framework defines the following components:

1. Subscription request

   A message sent to the notification source containing the location of the notification sink to which notification messages are to be sent, and an initial lifetime for the subscription source. A subscription request causes the creation of a grid service instance called a subscription.

2. Subscription expression

   An XML document that describes the rules and format of the notification message, such as the notification destination, and when it should be sent.

3. Subscription manager

   A subscription manager instance is created to manage the subscription information.

4. Notification source

   A grid service instance that sends notifications.

5. Notification message

   An actual callback notification message. Notifications can also be sent to post service data value modifications. So, in this case, notifications use a service data concept behind the scenes. When a service instance wants to receive a notification associated to a particular SDE, the service needs to be subscribed in order to be notified of subsequent changes to the target instance's service data.

6. Notification sink

   A grid service instance that receives notifications.

The developer may want to execute a method based on the state of the service or based on the value or contents of a service's Service Data Element. This can be accomplished by subscribing to the service and informing it that this client would like to be notified when a certain condition is met. For example, a user of a service may want to be notified if the file system on the node where the service is running no longer has sufficient space left to submit jobs, as seen in Figure 5-3 on page 70. The job submission may be set up in such a way as to run jobs continuously until there is insufficient disk space left to store the output of the job. This scenario is probably not a real world use of notification in grid services, since there are standard products that monitor the infrastructure, but it does illustrate the concepts. A more common example might be one where we ask a bank service to notify us if the account balance reaches a level that is below a certain number of dollars.

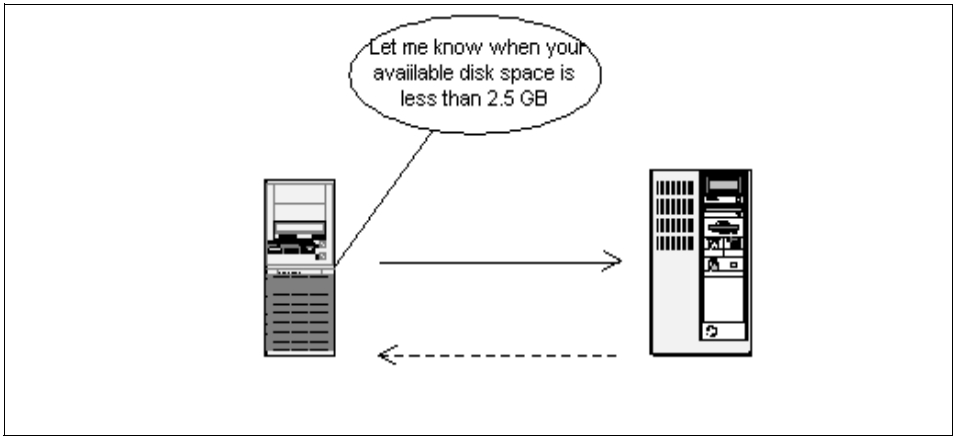


Figure 5-3 Notification if no sufficient space left to submit jobs

There are two basic implementation approaches to notifications. In the first case, called push notification, a client subscribes to be notified when a condition is met. When that condition is met, a notification is sent to the subscriber which also includes information about the condition which was met. In many cases, the notification is sent along with the value of the Service Data Element which was subscribed to, as shown in Figure 5-4.

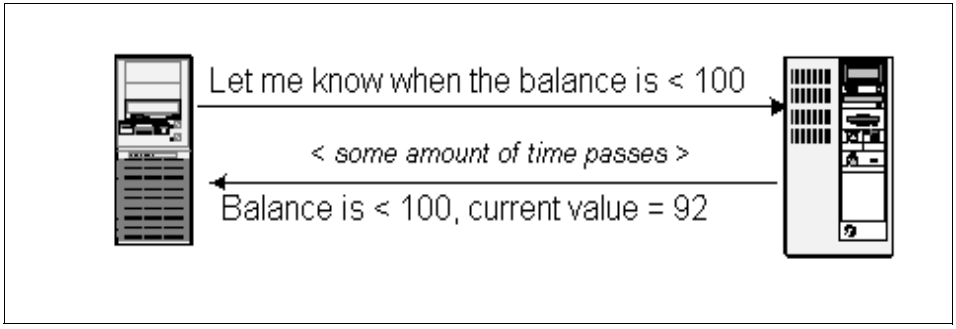


Figure 5-4 Push notification

It is quite common to subscribe to Service Data Elements and request to be notified when the value changes. However, not all clients who subscribe to the notification need to know the new value of the Service Data Element. In some cases, the client may just need to know that the value has changed and not necessarily what the current value is. For this scenario, another kind of notification, called pull notification, is more efficient. In the pull notification approach, the subscriber is notified of the change of state or value of the Service Data Element, but the new or updated value of the Service Data Element is not sent with the service notification. Subscribers who wish to know the current value

of the Service Data Element can make another call to the service requesting the current value of the Service Data Element (pull). This approach, as shown in Figure 5-5, is typically used when clients need to know that a change occurred but not necessarily the new value of the Service Data Element.

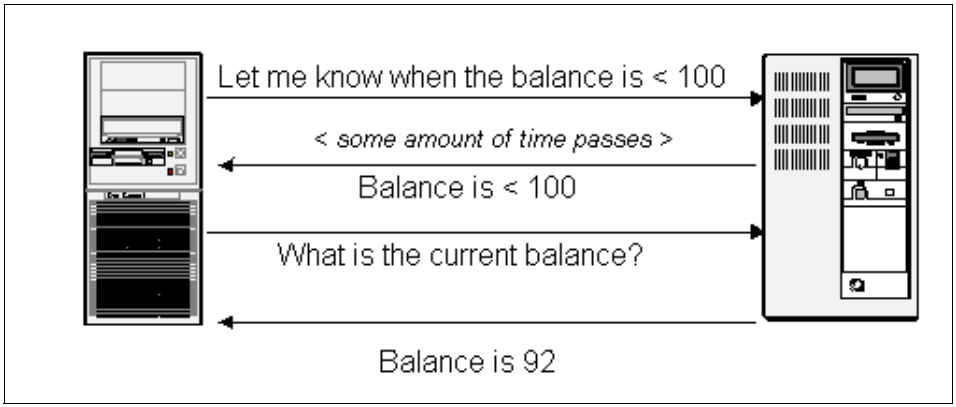


Figure 5-5 Notification and pull SDE

Multiple clients can subscribe to be notified in the event of a change of value in a Service Data Element. The service, in this case, is called the notification source. The subscribers or clients are called the notification sink. Multiple clients could subscribe to a service's Service Data Element as a way to share data. Notifications are handled by the subscription management service. The service or notification source notifies the subscription management service that the value has changed. The subscription management service maintains a list of clients or other services which are interested in the Service Data Element. The subscription management service then notifies each client or notification sink which has registered for notification of that Service Data Element. Note that when implementing notifications, the service should be non-transient and not a service that is created by a client to do some work, then be destroyed.

6

Project and design of grid applications

This chapter provides an overview of the issues to consider for any grid application. The approach to build a grid-enabled application encompasses a wide range of aspects of problem analysis, application architecture, and design. Some of these items may not apply for every project. Some aspects are familiar from other application development projects and are not elaborated upon in great detail. Others, which are new aspects due to the nature of a grid application, are examined in greater detail.

6.1 Use existing code or build from scratch?

We start our development by examining where the project is starting from and then perform a qualification analysis of the system to see if it makes sense to put it into a grid at all. Once we are satisfied, we move into the standard software project phases of requirements gathering, design, coding, testing and deploying.

The developer is usually in one of two project kick-off situations:

- Writing a new grid application
- Writing a grid “wrapper” around an existing running system

While most developers would like to create new, pure systems to have the most control over design and development, that is almost never the case. There is an enormous number of existing systems, which usually evolve over time within an organization. We briefly discuss the issues around a grid project’s starting point.

6.1.1 Developing a grid application from scratch

A newly specified grid system gives the developers a great deal of control:

- Freedom to choose programming language and tools
- Freedom to choose best-of-breed technologies and toolkits
- Blank sheet of paper for all design aspects

Developers in this situation will be able to easily follow the development steps outlined in this redbook, but they should remember that no system is an island and there will usually be constraints applied to their design, regardless of its nature as a grid application:

- Must handle connections to existing databases, message queues, network sockets
- Must read and write existing data formats
- Must comply with local security policy

6.1.2 Grid enabling existing code

Life is slightly more complicated for those developers whose project is to place an existing system into a grid environment. This task is generally referred to as “wrappering” the existing code and involves the following issues. These are meant to be high-level views of handling issues in existing code, but the next section serves as a checklist of items which can block successful development of a grid service. If difficulties arise in this section about the existing codebase, then the system has a strong chance of failing the qualification criteria in the next section.

Existing code is written in Java

This wrappering is easy to perform if the existing system is written in Java, since it requires writing a new Java class which exposes one or more public methods. As you will see later in the document, the Operation Provider method was designed for just this situation. The methods in the new class massage the parameters and perform whatever processing is necessary to call into the existing system's object structure and return any results.

Existing code is not written in Java

If the existing system is not written in Java, Java Native Interface (JNI) techniques can be used, but there is an immediate restriction on the operating system platform the grid service can be deployed into. A service calling directly into a Windows DLL can only be deployed onto Windows machines; Linux and other Unix platforms have the same restriction for their shared library files. This is not a big problem if you have a homogeneous grid, but the nature of most grids is a heterogeneous evolution over time and you will be limiting the deployment potential of your grid service.

Existing code is small and encapsulated

This is the best kind of existing code situation, since it is possible to build a single-task, well-defined service and deploy it around a grid. This service is typically a CPU-intensive process that accepts some input, performs some processing and returns the results when done.

Existing code is large and/or has multiple connections

If your existing code has these characteristics, then its candidacy as a grid service is already in doubt. Your grid nodes must be very powerful to handle large and complex tasks within the performance metrics defined by the requirements. If the code makes multiple connections to other systems and/or transfers a lot of data, the service is not going to parallelize well and you will end up with one or more bottlenecks, negating the benefits of running in a grid. It is critical to attempt a refactoring of the existing code to modularize it and remove as many interdependencies as possible, until you end up with a small and encapsulated solution.

6.2 Qualify the application

It is important to first qualify an application for its suitability for running on a grid. Not all applications lend themselves to successful or cost-effective deployment on a grid. A number of criteria may make it very difficult, require extensive work effort, or even prohibit grid-enabling an application. We provide a criteria list for analyzing an application. The list includes attributes of an application or its

requirements that may inhibit an application from being a good candidate for a grid environment. The list may not be complete and depends on the local circumstances of resources and infrastructure. A qualification scheme document that acts as a basis for architecture and project planning for a grid application can be found at http://developerWorks®.ibm.com/.

Some items such as temporary data spaces, data type conformity across all nodes within the network, appropriate number of software licenses available in the network for the grid application, higher bandwidth, or the degree of complexity of the job flow can be solved, but have to be addressed up front in order to create a reasonable grid application.

An application with a serial job flow can be submitted to a grid, but the benefits of grid computing may not be realized, and the application may be adversely affected due to grid management overhead. However, exploiting the grid and submitting the application to more powerful remote nodes may very well provide business value.

The following list presents the most critical items that hinder or exclude an application from use on a grid:

1. High inter-process communication between jobs without high speed switch connection (for example, MPI); in general, multi-threaded applications need to be checked for their need of inter-process communication.
2. Strict job scheduling requirements depending on data provisioning by uncontrolled data producers.
3. Unresolved obstacles to establish sufficient bandwidth on the network.
4. Strongly limiting system environment dependencies for the jobs.
5. Requirements for safe business transactions (commit and roll-back) via a grid. At the moment, there are no standards for transaction processing on grids.
6. High interdependencies between the jobs, which expose complex job flow management to the grid server and cause high rates of inter-process communication.
7. Unsupported network protocols used by jobs may be prohibited to perform their tasks due to firewall rules.

6.3 Understand the requirements

Once we understand the project's starting point and have qualified it as reasonable to implement as a grid service, we begin examining the project's requirements.

Requirements are key to the successful design, development and deployment of a system. As all programmers are aware, without a reasonably detailed set of requirements, no system can ever be delivered on time, on budget and with full functionality.

There are dozens of specific methodologies for requirements definition in use today. This document is not going to try to convince you to use one style or another; the important thing is for developers to use one with which they are comfortable, the key word being *use*. Projects which are under strong time constraints and pressure get to a point in the development cycle where the team must has something to “show”. The rule of thumb is that most projects require about 50% of the total effort to be spent in requirements analysis, architecture and design before the first line of code is written. Unless you have management that understands the key points of the software development process, you may have to defend why there is no code running yet. In this way, you can assure management that the project is on track and the necessary work is being done.

A methos that has been successful in the past involves defining all requirements and their corresponding use cases in the project documentation, then have all related parties physically sign their name to an acceptance letter. The signatories should include the project managers, the user representatives, the business representatives, the architects, the designers, the developers, the systems administrators and the testers. The act of committing oneself to the acceptance of a document by signing it will make almost every member of the project perform a much more careful review. Once everyone has signed, it can be reasonably assumed that the requirements meet the final system goals and there will be no surprises or scope creep. Any substantial changes during development are handled as Change Requests and are either accommodated by more budget and time or are deferred until the next version.

6.3.1 Functional requirements

Functional requirements capture the intended behavior of the system, in terms of the services the system is intended to provide. Functional requirements vary widely depending on the operation of the system, but some examples are:

- User login
- User logoff
- Customer Data Entry and Validation
- Transaction
- Print Monthly Report
- Overnight Batch Process
- Pulling Data from Business Partner

One very common method used to determine the completeness and correctness of a set of functional requirements is the use case. Use cases are a set of

step-by-step descriptions of process flows through a system with references to all internal and external components which are affected. One ore more use cases can map into a high-level requirement such as those defined above. A simple system could get by with a handful of use cases while a complex system could easily have hundreds or even thousands. Use cases can be used by the test team to verify that the requirements were met. When all test cases are successful, the system can be considered both feature complete and accurately implemented.

Use cases are generally described in the Unified Modeling Language (UML), defined by Grady Booch, Jim Rumbaugh and Ivar Jacobson of Rational® Software Corporation (now IBM) and now shepherded by the Object Management Group. In addition to Use Case diagrams, UML has additional object-oriented design and development capabilities, including:

- Class Diagrams: class definitions and relationships to other classes. Most UML tools can generate Java code directly from these diagrams.
- Interaction Diagrams: time sequence of objects in the system
- State Diagrams: sequences of state that an object passes through in its life cycle
- Activity Diagrams: a special class of State Diagram

Since UML has become the predominant OO modeling methodology, there are many products that allow architects and developers to build a full model of the system, then generate Java code directly from the model. IBM's Rational Rose® family of products provides a complete suite of UML tools.

6.3.2 Non-functional requirements

Non-functional requirements define the softer goals of the project that are not defined in use cases, such as:

- Scalability
- User interface factors
- Error handling
- Security
- Application flexibility
- Server and client platform
- External connections
- Performance
- Reliability
- System management
- Topology considerations

- Mixed platform environments
- File formats
- Software license considerations

Scalability

As more work is sent to a system, utilization of resources increases until the system cannot handle any additional load. At this point, the system can fail in any number of ways, including system crashes, lost messages, corrupted data, user request time-outs, etc. The goal of a well-designed system is to be able to seamlessly add more resources to the system to handle additional load, while providing near-linear performance increases and requiring no system redesign.

The application and its server environment should be specified to allow its deployment on nodes around the grid. In many cases, it is possible to have multiple server instances on a single node without significant system degradation. In fact, the goal is usually to run the nodes at a high utilization factor, reducing the overall system cost. When a node's resources are exhausted, additional nodes can be added, each running multiple instances.

User interface factors

Great strides have been made over the last few years in terms of GUI interface design, but programmers tend to build user interfaces that provide the needed functionality and do not always think about other factors, such as:

- Overall screen layout: the GUI application window paradigm with a menu bar across the top, a button bar underneath, a scroll bar on the right side and perhaps across the bottom, with the application's data content in the center is the de facto standard across every modern operating system.
- Menus: the classic File -> Edit -> View -> ... -> Help menu structure is well understood. Buttons: users are used to the OK, Cancel, and Help button text and placement in dialogs.
- Accelerator keys: key combinations such as Alt-key and Ctrl-key bound to menu items and dialog buttons allow mouseless operation of the application. Some users find this preferable to constantly moving their hands between the keyboard and mouse and it takes almost no effort by the developer.
- Multi-dialog "wizards": when collecting complex or lengthy information from the user, a sequence of dialogs can be presented. Each dialog focuses on a single part of the whole, allowing the user to focus on that aspect before moving on. A Previous button is essential to allow the user to back up in the process to correct any errors before continuing.
- Context-sensitive Help: most screens and dialogs have associated help pop-ups, but increasingly users are able to click specific items and get fine-grained help on just that item. It is also imperative to ensure that the

provided help is actually helpful, allowing the user to understand a concept, solve a problem or be taken to an external source for additional help.

- Internationalization: most applications will eventually have users with different native languages and it will satisfy those users and perhaps even open additional markets for the application if users can work in their native language. This requires some additional thought and work in the application for such items as:
  - Dynamically loading all text strings from external language bundles, properly merging application values as needed.
  - Ensuring enough screen real estate is reserved for the longest possible native language string; for the most common supported languages, this is usually the German version.
  - Performing quality translations of the primary language bundle. There are companies that specialize in doing just this type of work and they are very familiar with the technical aspects of various types of translation files.
- Disabled access: ensuring that nothing is done in the user interface that will prevent a disabled user from effectively using the application. This can include:
  - Providing alternate text for graphical screen items such as icons, for speech programs and braille keyboards
  - Avoiding complicated multi-key sequences for hand/finger disabled users
  - Providing complete functioning of the application by keyboard only for mouse-disabled users

The overall goal is to allow users to come up to speed quickly and effectively use the application. By following accepted UI factors and putting in a small amount of additional effort, you will have a much happier user community.

Error handling

There is little which is more frustrating to a user than when an error occurs and is not handled gracefully by the application. It is very easy for developers to perform little or no error checking and error handling within the application. Hopefully, comprehensive unit and system testing will discover these deficiencies in the code, but beyond that, a consistent processes for catching errors in code, logging them, and displaying them to the user should be defined. Java's exception mechanism makes it natural to generate and handle application errors, but the developers need a defined process to follow to determine:

- When to throw an exception
- When to catch an exception
- When to re-throw an exception
- When to log the exception

- When to try to handle the exception via retries, re-interpreting data, etc.
- When and how to display an error to the user
- What option to give to the user
- What to do when the user makes a choice

Security

Security is a complex area and is therefore sometimes not handled to the proper extent within an application. Developers must take the following into account to provide a secure environment:

- User Authentication: users are forced to log in to the system before any sensitive data or operations are available. Authentication involves the user providing some sufficient number of authentication tokens, including a user ID, a password, a smartcard, a time-dependent code such as that used by the RSA SecurID (see http://www.rsasecurity.com/products/securid), or in extreme cases, a biometric factor such as a retina scan or fingerprint.
- User Authorization: authorization schemes allow the user to only perform actions they have been explicitly allowed to do. This can be handled either programmatically in the application or by delegating the decision to an authorization system such as from IBM's Tivoli family of security products.
- Data encryption: sensitive information should never cross the company firewall without being encrypted and even certain information, such as user authentication tokens, flowing inside a company should be encrypted. Luckily, the world of data encryption is very mature, with strong standards in place, generally based on PKI (Public Key Infrastructure). Web browsers can securely connect to Web servers using SSL/TLS encryption and the browser and server automatically negotiate to an acceptable level of encryption. As a caveat, a Web server basic authentication challenge based on sending the browser a '401' page returns the user ID and password in a Bas64 encoding which could be very easily turned back into clear text and used by a malicious user.
- Logging and Alerting: a guideline for developers to follow when important events occur. Real-time or later analysis of these logs can be very useful for detecting repeated application failure, attempted attacks on the system, etc.

Application flexibility

A system is almost never written, deployed and never touched again. When the inevitable bug fixes, performance fixes or new feature requests come along, the system should be structured with a view to the future so these issues can be handled without breaking interfaces or causing significant rewrites. Items such as these can help avoid problems in the future:

- Separation of code and data: do not bind the data type objects too tightly to the code. This can be done by adding a layer of indirection on top of the real

data objects, allowing virtually the entire data structure of the application to be changed with no change to the application code

- Definition of published interfaces: this is similar to the layer of data indirection, but is designed to sit between distinct areas of the application code, forming the object and method contract. The real application code can be changed completely without having any effect on the external modules
- Plug-in design: define the system as a set of independent modules that plug in to the base either by having an entry point class listed in a configuration file or by dynamic discovery of classes which implement a particular application plug-in interface. This allows new modules to be added to the base system with no impact on the existing code base. This takes a great deal of forethought and design, but the time is well invested if the system is expected to grow over time.

Server and client platform

Everyone involved in the system should understand what the minimum and recommended hardware platforms are, as well as the list of software prerequisites so there are no surprises during development, testing and user execution.

One common example is a Web-based system that expects the user to use a Web browser. Generally, all is well if the Web designers develop only Web pages that adhere to the W3C's HTML standards. The W3C has a number of validations suites on their system that generate reports on how closely the Web page conforms to the standards. These suites can be found at:

`http://www.w3c.org/QA/Tools/#validators`

Browsers such as Mozilla, Opera, Konquerer and Safari all work quite well with valid content. Projects go wrong when they introduce browser-specific content into their Web pages, either by using non-standard tags or by doing JavaScript testing of the user's browser and providing specific content tailored to that browser. A few years ago, this was a common issue because the browsers (generally Netscape and Internet Explorer) had difficulty with some valid content. These days, there is generally no reason to resort to such measures.

Unfortunately, due to the dominance of Internet Explorer, some Web developers (as a side effect of using certain Web development tools) build their sites with Internet Explorer-specific content, forcing users to use Internet Explorer to successfully access the site. The lesson is: build Web sites with fully W3C validated content and all users will be able to access the site with their choice of browser.

External connections

Almost every application requires a connection to external systems to read and/or write data. Before development begins, it is critical to list all external systems, along with their communication methods, data packet formats, authentication, authorization, connection method, recovery procedure, expected performance and human contact names. Once the staff on both ends of the connection agree on all aspects, development can begin and it usually goes much more smoothly than if the data exchange is defined poorly or incorrectly; in that case, the teams will have a great deal of difficulty finishing the testing of the systems.

Performance

When considering enabling an application to execute in a grid environment, the performance of the grid and the performance requirements of the application must be considered. The service requester is interested in a quality of service that includes acceptable turnaround time. Of course, if the project is building a grid and one or more applications that will be provided as a service on the grid, then the service provider also has interest in maximizing the utilization and throughput of the systems within the grid. The performance objectives of these two perspectives are discussed below.

Resource provider's perspective

The performance objective for a grid infrastructure is to achieve maximum utilization of the various resources within the grid to achieve maximum throughput. The resources may include but are not limited to CPU cycles, memory, disk space, federated databases, or application processing. Workload balancing and preemptive scheduling may be used to achieve the performance objectives.

Applications may be allowed to take advantage of multiple resources by dividing the grid into smaller instances to have the work distributed throughout the grid. The goal is to take advantage of the grid as a whole to improve the application performance.

Service requester's perspective

The turnaround time of an application running on the grid could vary depending on the type of grid resource used and the resource provider's quality-of-service agreement. This assumes that the job is started immediately and that it is not preempted by another job during execution. The same batch job may be scheduled to run overnight when the resource demands are lower if a quick turnaround is not required. The resource provider may charge different prices for these two types of service.

If the application has many independent sub-jobs that can be scheduled for parallel execution, the turnaround time could be improved appreciably by running each sub-job on multiple grid hosts.

Turnaround time factors

There are some factors that can impact the turnaround time of applications run on the grid resources. For example, these could include the following.

Communication delays

Network speed and network latency can have a significant impact to the application performance if it requires communicating with another application running on a remote machine. It is important to consider the proximity of the communicating applications to one another and the network speed and latency.

Data access delays

The network bandwidth and speed will be the critical factors for applications that need to access remote data. Proximity of the application to the data and the network capacity/speed will be important considerations.

Lack of optimization of the application to the grid resource

Optimum application performance is usually achieved by proper tuning and optimization on a particular operating system and hardware configuration. This poses possible issues if an application is simply loaded on a new grid host and run. This issue may be resolved if the service provider makes an arrangement with the resource provider so that the application's optimum configuration and resource requirements are identified ahead of time and applied when the application is run.

Contention for resource

Resource contention is always a problem when resources are shared. If resource contention impacts performance significantly, alternate resources may need to be introduced. For example, if a database is the source of contention, then introducing a replica may be an answer. In addition, the network may need to be divided to separate the traffic to the databases. Optimum sharing of the grid hosts may be achieved by a proper scheduling algorithm and workload balancing. For example, the shortest job first (SJF) batch job scheduling algorithm may provide the best turnaround time.

Network reliability

Failures in the grid resource and network can cause unforeseen delays. To provide reliable job execution, the grid resource may apply various recovery methods for different failures. For example, in the checkpoint-restart environment, some amount of delay will be incurred each time a checkpoint is taken. A much longer delay may be experienced if the server crashed and the

application was migrated to a new server to complete the run. In other instances, the delay may take the entire time to recover from a failure such as network outages.

We talk about this in the following paragraphs.

Reliability

Reliability is always an issue in computing, and the grid environment is no exception. The best method of approaching this difficult issue is to anticipate all possible failures and provide a means to handle them. The best reliability is to be surprise tolerant. The grid computing infrastructure must deal with host interruptions and network interruptions. Below are some approaches to dealing with such interruptions.

Checkpoint-restart

While a job is running, checkpoint images are taken at regular intervals. A checkpoint contains a snapshot of the job states. If a machine crashes or fails during the job execution, the job can be restarted on a new machine using the most recent checkpoint image. In this way, a long-running job that runs for months or even years can continue to run even though computers fail occasionally.

Persistent storage

The relevant state of each submitted job is stored in persistent storage by a grid manager to protect against local machine failure. When the local machine is restarted after a failure, the stored job information is retrieved. The connection to the job manager is reestablished.

Heartbeat monitoring

In a healthy heartbeat, a probing message is sent to a process and the process responds. If the process fails to respond, an alternate process may be probed.

The alternate process can help to determine the status of the first process, and even restart it. However, if the alternate process also fails to respond then we assume that either the host machine has crashed or the network has failed. In this case, the client must wait until the communication can be reestablished.

System management

Any design will require a basic set of systems management tools to help determine availability and performance within the grid. A design without these tools is limited in how much support and information can be given about the health of the grid infrastructure. Alternate networks within a grid architecture can be dedicated to perform these functions so as not to hamper the performance of the grid.

Topology considerations

The distributed nature of grid computing makes spanning across geographies and organizations inevitable. As an intra-grid topology is extended to an inter-grid topology, the complexity increases. For example, the non-functional and operational requirements such as security, directory services, reliability, and performance become more complex. These considerations are discussed briefly in the following paragraphs.

Network topology

The network topology within the grid architecture can take on many different shapes. The networking components can represent the LAN or campus connectivity, or even WAN communication between the grid networks. The network's responsibility is to provide adequate bandwidth for any of the grid systems. Like many other components within the infrastructure, the networking can be customized to provide higher levels of availability, performance, or security.

Grid systems, are for the most part, network-intensive due to security and other architectural limitations. For data grids in particular, which may have storage resources spread across the enterprise network, an infrastructure that is designed to handle a significant network load is critical to ensuring adequate performance.

The application enablement considerations should include strategies to minimize network communication and network latency. Assuming the application has been designed with minimal network communication, there are a number of ways to minimize the network latency. For example, a gigabit Ethernet LAN could be used to support high-speed clustering or utilize high-speed Internet backbone between remote networks.

Data topology

It would be desirable to assign executing jobs to machines nearest to the data that these jobs require. This would reduce network traffic and possibly reduce scalability limits.

Data requires storage space. The storage possibilities are endless within a grid design. The storage needs to be secured, backed up, managed, and/or replicated. Within a grid design, you want to make sure that your data is always available to the resources that need it. Besides the issue of availability, you also want to make sure that your data is properly secured, as you would not want unauthorized access to sensitive data. Lastly, you want more than decent performance for access to your data. Obviously, some of this depends on the bandwidth and distance to the data, but you will not want any I/O problems to slow down your grid applications. For applications that are more disk intensive,

or for a data grid, more emphasis can be placed on storage resources, such as those providing higher capacity, redundancy, or fault-tolerance.

Mixed platform environments

A grid environment is a collection of heterogeneous hosts with various operating systems and software stacks. To execute an application, the grid infrastructure needs to know the application's prerequisites to find the matching grid host environment. Below are some things that the grid infrastructure must be aware of to ensure that applications can execute properly. It is equally important for the application developer to consider these factors in order to maximize the kinds and numbers of environments in which the application will be able to execute.

Runtime considerations

The application's runtime requirements and the grid host's runtime environments must match. As an example, below are some considerations for Java applications. Similar requirements may exist for applications developed in other languages.

Java Virtual Machine (JVM)

Applications written in the Java programming language require the Java Virtual Machine (JVM). Java applications may be sensitive to the JVM version. To address this sensitivity, the application needs to identify the JVM version as a prerequisite. The prerequisite may specify the required JVM version or the minimum JVM version.

Java applications may be sensitive to the Java heap size. The Java application needs to specify the minimum heap size as part of its prerequisite.

Java packages such as J2SE or J2EE may also need to be identified as part of the prerequisites.

Availability of application across platforms (portability)

The executables of certain applications are platform-specific. For example, an application written in the C or C++ programming language needs to be recompiled on the target platform before it can be run. The application could be pre-compiled for each platform and the resulting executables marked for a target platform. This will increase the number of qualifying grid host environments where the application can run. The limitation of this method will be the cost-effectiveness of porting the application to another platform.

Awareness of OS environment

The grid is a collection of heterogeneous computing resources. If the application has certain dependencies or requirements specific to the operating system, the

application needs to verify that the correct environment is available and handle issues related to the differing environments.

File formats

Knowledge of the format of files in the system is necessary when the output of an application running on one grid host is accessed by another application running on a different grid host. The two grid hosts may have different platform environments. XML may be considered as the data exchange format. XML has now become popular not only as a markup language for data exchange, but also as a data format for semi-structured data.

Software license considerations

One question that commonly arises when discussing grid computing is that of software license management. There are many products and solution designs that can help with license management.

Commercial software licenses

It is important to discuss how to deal with software licenses that are used inside the grid. Insufficient numbers of licenses may seriously hinder the expansion or even exclude certain programs or applications from being used in a grid environment.

The latter is the case if the grid is going to access personally licensed applications on a personal computer, for example, in a scavenging mode use of single-user licensed software. This cannot be done without violation of the license agreement.

Different models

The range of license models for commercial software spans from all restrictive to all permissive.

Between these two extremes, there are numerous models in the middle ground, where licenses are linked to a named user (personal license), a workgroup, a single server, or a certain number of CPUs in a cluster, to a server farm, or linked to a certain maximum number of concurrent users.

Software licenses are given with a one-time charge or on a monthly license fee base. They can include updates or require the purchase of new licenses. All this varies from vendor to vendor, and from customer situation to customer situation, depending on individual agreements or other criteria.

Software licenses may allow for the migration of software from one server to another or may be strictly bound to a certain CPU. Listing all possible software licensing models could easily fill a book, but we cover a few next.

Service provider license agreement

Subscriber Access Licenses (SALs) are offered by service providers, for example, on a pay-per-use basis or as a flat rate for a certain maximum number of access times per month/week/year.

IT service providers, in turn, may acquire software licenses from ISVs for use by their customers, or they may simply host software for which the end user will pay directly to the providing ISV according to their agreed license model.

Open source licensing

Another complexity is added when a software product is built that contains or requires open source software like the Globus Toolkit or Apache. The open source model is based on the principle that anybody (an ISV or private person) can provide software to any interested party that can be modified, customized, or improved by the recipient.

The modifying recipient, in turn, can offer this changed code to anybody, who again can change it when needed. Therefore, there can be many developers in a loose community participating in development and improvement of a given set of code.

In this case, licenses are not bound to binary executables but cover source code as well. The following three licensing models for open source software are the most common, though there are several more, which may need to be investigated in a specific case.

FreeBSD, MIT, Apache (all permissive licenses)

The license models for FreeBSD, MIT, and Apache are all permissive, which means that they allow for free distribution, modification, and license changes. Software without copyright (public domain software) falls under this category as well.

For details on FreeBSD, see:

`http://www.freebsd.org/`

For details on BSD licenses, see:

`http://www.opensource.org/licenses/bsd-license.php`

For MIT licenses, see:

`http://www.opensource.org/licenses/mit-license.php`

For the Apache Software License, see:

`http://www.opensource.org/licenses/apachepl.php`

LGPL (persistent license)

The Lesser General Public License (LGPL) allows free distribution of the software, but restricts modifying it. All derivative work must be under the same LGPL or GPL. The definition of this license type can be found at:

http://www.opensource.org/licenses/lgpl-license.php

GNU GPL, IBM Public License (persistent and viral license)

The GNU General Public License (GPL) as well the IBM Public License (PL) shows a persistent and viral model, which means that it allows free distribution and modifying, but all bundled and derivative work must be under GNU GPL as well.

The GNU GPL can be found at either of the following Web sites:

http://www.gnu.org/copyleft/gpl.html
http://opensource.org/licenses/gpl-license.php

The IBM PL can be found at:

http://www.opensource.org/licenses/ibmpl.php

For Open Source Initiative (OSI) certified licenses and approvals, visit:

http://opensource.org/docs/certification_mark.php

For the OSI portal, simply go to:

http://www.opensource.org

There is a list of all approved open source licenses at the following Web site. GPL, LGPL, BSD and MIT are the most commonly used so-called "classic" licenses.

http://www.opensource.org/licenses/

License management tools

In order to manage most of these license models in a network, there are a number of license management tools available. These tools ensure that all software that is included in a network or a grid application is properly used according to its license agreements.

Most of the license manager providers offer an SDK with APIs for various programming languages. The span of license models covered by each product varies. In the following sections, some of the most often used tools are listed.

FLEXlm

In the Linux world, FLEXlm is foremost; it offers 11 core models and 11 advanced licensing models. The core models include: Node-locked, named-user, package,

floating (concurrent) over network, time-lined, demo, enable/disable product, upgrade versions, etc.

The advanced licensing models span from capacity, over site license, license sharing (user, groups, hosts), floating over list of hosts, high-water mark, linger license, overdraft, and pay-per-use, to network segments and more.

The complete list of supported licensing can be found at the following Web site:

http://www.globetrotter.com/flexlm/lmmodels.sthm

More information about the use and advantages of this de-facto standard of electronic license management technology in the Linux world is available at:

http://www.globetrotter.com/flexlm/flexlm.shtm

Tivoli License Manager

IBM Tivoli License Manager is a software product that supports management of licenses in a network. Due to its nature, it is possible to reflect most of the license models being used in the industry. IBM Tivoli License Manager can reflect various stages of use during a piece of software's life time.

The IBM Redbook *Introducing IBM Tivoli License Manager*, SG24-6888, provides examples of how to reflect IBM, Microsoft®, Oracle, and other vendors' license models in its management.

IBM Tivoli License Manager is integrated with WebSphere Application Server and available for AIX, Solaris, and several Microsoft Windows platforms.

More details about the product are also given on the IBM Software Group Web site at:

http://www.ibm.com/software/tivoli/products/license-mgr/

IBM License Use Management (LUM)

IBM License Use Management (LUM) in its current version (4.6.6) is designed for technical software license management since it is deployed by most IBM use-based software products. It is intended to be integrated with any vendor software in order to control use-based licensing of the software.

LUM is available for all Windows platforms, AIX, HP-UX, Linux, IRIX, and Solaris. It supports a wide range of C, C++, and Java development environments. It can be used in networks with most of the available Web servers.

Software developers can reflect various use-based license models while integrating LUM APIs in their software products. It can be used for monitoring and controlling the use of software in networks.

More details can be found on the IBM software group Web site at:

http://www.ibm.com/software/is/lum/

Platform Global License Broker

Among the various ISVs that offer grid software products, Platform shows a special grid-oriented license management feature named Platform Global License Broker.

This product runs on AIX, HP-UX, Compaq Alpha, and IRIX. It uses Globetrotter FLEXlm 7.1 as described above. More details on Platform Global License Broker are available on the Internet at:

http://www.platform.com/products/wm/glb/index.asp

General license management considerations

When designing and deploying grid-enabled applications, it is important to understand any licensing requirements for required runtime modules. If designing a broker or utilizing MDS to identify possible target resources on which to run the application, the existence or applicability of any required software licenses should be taken into account.

6.4 Develop a high-level design

Once the requirements have been defined and signed off, a high-level design can be finalized. You will likely find the architects building prospective high-level designs during the requirements phase. Now those designs must be fleshed out, evaluated and finalized.

When building a grid system and having the GT3 as a requirement, the design should try to exploit as much of the GT3 functionality as possible. There is no sense in reinventing the wheel when the GT3 toolkit provides rich function, already in use by other designers around the world. You simply layer your application-level problem on top of the GT3 features and fill in any remaining holes of function.

GT3 is a large toolkit and there are features that we do not describe in this document. If you have a sufficiently complex application, it will be a valuable use of your time to read the GT3 documentation after finishing this document to see what else is available before you begin your design. The feature you need may be available, saving you a lot of design, development and testing time.

In the grid arena, the first part of the high-level design is the definition of the interface contract that the services make to the rest of the grid.

6.4.1 Define interfaces

The grid service interface is all of the public aspects that you will be providing, including methods (with their parameters and return types), service data and notification strategy. The definition of a grid service is very similar to the definition of a Web service, but grid service developers have a few more features to use.

6.4.2 Define method parameters and return types

Each grid service is comprised of one or more methods. In the code for the client of a service, with the help of a small number of Globus helper classes, a local proxy object is created that represents the running grid service somewhere else on the network. The methods that are defined as public by the remote service are by nature also available on this local proxy object.

In addition to the list of method names, the designer must define the full method signature, including any passed-in parameters and any returned values.

Programmers can define the grid service's interface in one of two ways:

- **Top-down approach**: a grid service description definition is written in an XML language called GWSDL, which is an extension to the Web services WSDL language.
- **Bottom-up approach**: A Java Interface class is created which defines all method signatures in native Java syntax.

Later in the document, we go over the details and discuss the pros and cons of each.

6.4.3 Define service data and notification strategy

As discussed previously, grid services are written as an extension to the Web services standards. Everything we have discussed so far in this chapter is compliant with Web services. Two of the first major extensions that move designers into the grid services world are service data and notification.

Service data

SDE (5.3, "Service Data Elements" on page 61) is a collection of objects which are directly accessible by clients. This differs from method-based access to data since the object itself is public and any client can access it simply by requesting it by name. The service interface can place restrictions on service data by defining attributes such as the minimum and maximum number of instances of the SDE, whether clients can write to the object (in addition to the default of only having read access) and whether an object must always exist or can be null.

Notification

Service Data and notification (see 5.5, “Notifications” on page 67) go virtually hand in hand. As described above, service data can be used to implement asynchronous processing, but how does the client know that the operation is complete and the data is waiting in an SDE? Notification is the answer. Notification operates on a subscription model; when a client wants notification against an SDE, it uses GT3 helper methods to subscribe, given the SDE’s symbolic name. When the grid service places a result in an SDE, it uses GT3 helper methods to fire notifications to all of the client subscribers. The notification can be of one of two styles:

- Push: the SDE in question is sent along with the notification event to the client
- Pull: a dummy SDE is sent along with the notification event to the client, and the client has to query back to the SDE to get the updated value

The designers pick one or the other style depending on the needs of the application.

To summarize, Service Data and Notification from the designer’s point of view:

- Determine what data you want to expose in a non-method-call manner. Service data is generally used to expose the state of a service, intermediate calculations, percent completion, etc. These can also be exposed via method calls, but may make more sense as SDEs in a particular application.
- Determine if you will provide notification on one or more SDEs and whether you will use push or pull style. Notification is generally used to provide asynchronous notification of the change in state of an SDE. There is no way to provide this type of functionality short of a long-blocking method call or polling.

6.4.4 Define the life cycle

The life cycle of a grid service (see 5.4, “Life cycle” on page 64) starts with its instantiation, continues with its execution, and ends with its termination.

There are several ways to instantiate a grid service:

- Command-line tools provided by the GT3 toolkit: Helper commands such as **`ogsi-create-service`** are provided and can be used to instantiate a grid service. This is a useful tool during development and testing but in production, it is better to control in an automated manner rather than by human command.
- Automatic startup by grid service container: a grid service can be marked for automatic startup when you start the grid service container. When the container is up and running, your instance is immediately available.

- Programmatic creation using GT3 helper classes: the most common method is to use GT3 helper classes in an administration tool or in the service client to create instances.

At the time of creation, the new instance receives several calls to well-defined method names, that the instance can use to initialize itself. It could pull in data, establish connections with other parts of the system, create other service instances, etc.

During execution, the service instance responds to method calls, updates its SDEs and fires notifications as necessary.

There are several ways to terminate a service:

- Command-line tools provided by the GT3 toolkit: helper commands such as **`ogsi-destroy-service`** are provided and can be used to terminate a grid service. This is a useful tool during development and testing but in production, it is better to control in an automated manner rather than by human command.
- Automatic termination by grid service container: a grid instance defaults to having a lifespan of `Infinity`, but can be given a shorter lifespan at the time of creation. When that time expires, the grid service container gracefully terminates the instance. Note that any clients attempting to access the service afterward will receive exceptions.
- Programmatic destruction using GT3 helper classes: the most common method is to use GT3 helper classes in an administration tool or in the service client to destroy instances when they are no longer needed.

At the time of destruction, the instance receives several calls to well-defined method names that the instance can use to gracefully terminate itself. It could write out data, shut down connections with other parts of the system, destroy other service instances, etc.

6.4.5 Define security

Security is a very wide-ranging topic, but it is sufficient at this point of the design to simply note those components which have a security sensitivity. This could mean data that requires encryption, users who must pass an authentication phase, methods which must be checked for authorization, etc.

6.4.6 Run the scenarios to ensure that the requirements are satisfied

At this time, the architecture should be checked against the functional and non-functional requirements and the use cases to ensure that there are no problems.

6.5 Develop a detailed design

At this point, we take the high-level design and flesh out the details to a point where work can be handed off to developers. This stage can take quite a bit of time to ensure all details of the design are sufficiently defined. All system interfaces are fully specified, major classes are fully specified and other required behavior is defined.

Performing the detailed design requires a deeper linking between application constructs and the GT3 toolkit functionality. You have enough knowledge now to follow the document until the detailed GT3 features are explained below, just before we begin development of our sample application, but the one item we will cover now is how a grid application can behave, or flow, as part of solving a larger problem in a grid environment. Figuring out the best grid design and its implications for your application early will prevent possible problems if it turns out the design is not appropriate to the solution.

6.5.1 Application flow in a grid

First we define some terms:

Grid application: A collection of work items to solve a certain problem or to achieve desired results using a grid infrastructure. A grid application may consist of a number of jobs that together fulfill the whole task.

Job: Considered as a single unit of work within a grid application. It is typically submitted for execution on the grid, has defined input and output data, and execution requirements in order to complete its task. A single job can launch one or many processes on a specified node.

Data producer and consumer: Jobs that produce output data are called producers, and jobs receiving input data are called consumers. Instead of an active job as the final consumer of data, there can be a defined data sink of any kind within the grid application. This could be a database record, a data file, or a message queue that consumes the data.

In this chapter and in the light of these simple concepts, we will discuss the different application flows and the criteria that apply to jobs, data and others regarding usability and those that can be qualified as non-functional.

A grid-enabled application may consist of multiple jobs. Traditional applications execute in a well known and somewhat static environment with fixed assets. We need to look at the considerations (and value) for having an application run in a grid environment where resources are dynamically allocated based on actual needs.

If taking advantage of multiple resources concurrently in a grid, you must consider whether the processing of the data can happen in parallel tasks or whether it must be serialized and the consequences of one job waiting for input data from another job. What may result is a network of processes that comprise the application.

Application flow vs. job flow

We understand here that an application flow is the flow of work between the jobs that make up the grid application. The internal flow of work within a job itself is called the job flow. There are three basic types of application flows that can be identified:

- Parallel
- Serial flow
- Networked

We will discuss each of these in more detail in the following sections.

Parallel flow

If an application consists of several jobs that can all be executed in parallel, a grid may be very suitable for effective execution on dedicated nodes, especially in the case when there is no (or a very limited) exchange of data among the jobs.

From an initial job, a number of jobs are launched to execute on pre-selected or dynamically assigned nodes within the grid. Each job may receive a discrete set of data, fulfill its computational task independently and deliver its output.

The output is collected by a final job or stored in a defined data store. Grid services, such as a broker and/or scheduler, may be used to launch each job at the best time and place within the grid.

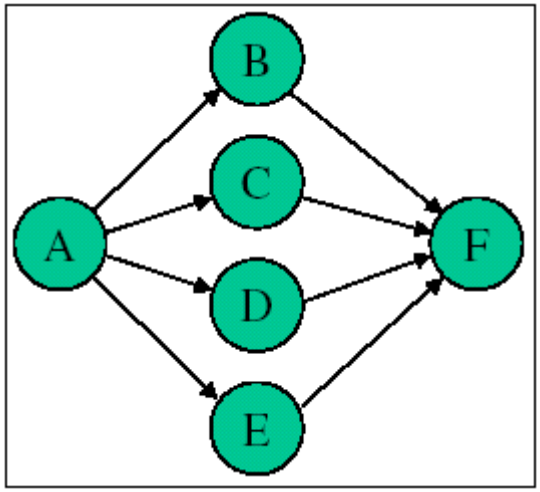


Figure 6-1 Parallel application flow

For a given problem or application, it would be necessary to break it down into independent units. To take advantage of parallel execution in a grid, it is important to analyze tasks within an application to determine whether they can be broken down into individual and atomic units of work that can be run as individual jobs.

This parallel application flow type is well suited for deployment on a grid.

Significantly, this type of flow can occur when there are separate data sets per job and none of the jobs need result from another job as input.

Serial flow

In contrast to the parallel flow is the serial application flow. In this case, as is shown in Figure 6-2, there is a single thread of job execution where each of the subsequent jobs has to wait for its predecessor to end and deliver output data as input to the next job. This means that any job is a consumer of its predecessor, the data producer.

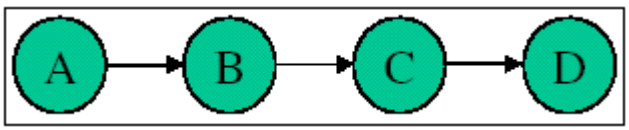


Figure 6-2 Serial job flow

In this case, the advantages of running in a grid environment are not based on access to multiple systems in parallel, but rather on the ability to use any of several appropriate and available resources. Note that each job does not necessarily have to run on the same resource, so if a particular job requires specialized resources, this can be accommodated, while the other jobs may run on more standard and inexpensive resources.

The ability for the jobs to run on any of a number of resources also increases the application's availability and reliability. In addition, it may make the application inherently scalable through the ability to utilize larger and faster resources at any particular point in time.

Nevertheless, when encountering such a situation, it may be worthwhile to check whether the single jobs are really dependent on each other, or whether, due to their nature, they can be split into parallel executable units for submission on a grid.

Parallelization

Section 2.1 of the redbook *Introduction to Grid Computing with Globus*, SG24-6895 provides certain thoughts about parallelization of jobs for grids. For example, when dealing with mathematical calculations, the commutative and associative laws can be exploited.

In iterative scenarios (for example, convergent approximation calculations) where the output of one job is required as input to the next job of the same kind, a serial job flow is required to reach the desired result. For best performance these kinds of processes might be executed on a single CPU or cluster, though performance is not always the primary criterion. Cost and other factors must also be considered, and once a grid environment is constructed, such a job may be more cost effective when run on a grid than by utilizing a dedicated cluster.

In case it is not possible to completely convert a serial application flow into a parallel one, a networked application flow may result.

Networked flow

In this case (perhaps the most common situation), complexity comes into play. Certain jobs within the application are executable in parallel, but there are inter-dependencies between them.

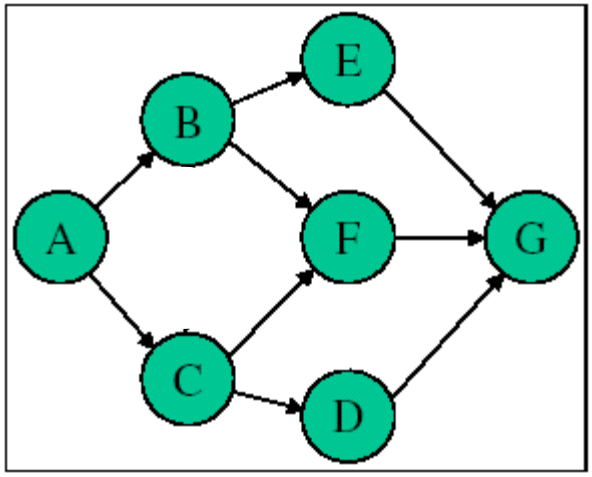


Figure 6-3 Networked job flow

Loose coupling

For a grid, this means the need for a job flow management service to handle the synchronization of the individual results. Loose coupling between the jobs avoids high inter-process communication and reduces overhead in the grid.

For such an application, you will need to perform more analysis to determine how best to split the application into individual jobs, maximizing parallelism. This also adds more dependencies on the grid infrastructure services such as schedulers and brokers, but once that infrastructure is in place, the application can benefit from the flexibility and utilization of the virtualized computing environment.

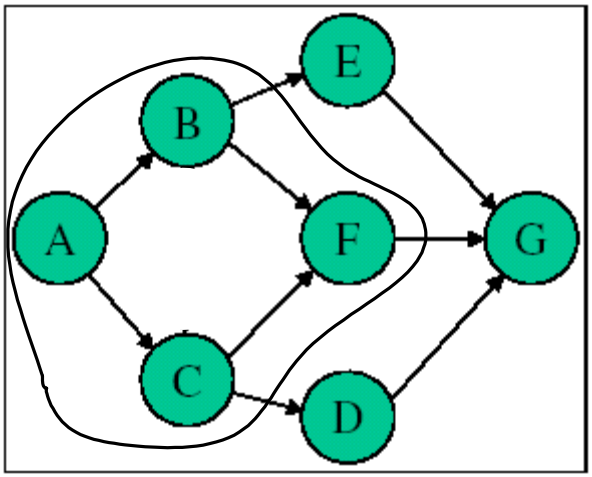


Figure 6-4 Jobs with sub jobs in a grid application

Jobs and sub-jobs

Another approach to ease the management of jobs within a grid application is to introduce a hierarchical system of sub-jobs. A job could utilize the services of the grid environment to launch one or more sub-jobs. For this kind of environment, an application would be partitioned and designed in such a way that the higher-level jobs could include the logic to obtain resources and launch sub-jobs in whatever way is most optimal for the task at hand. This may provide some benefits for very large applications to isolate and pass the control and management of certain tasks to the individual components.

6.5.2 Job criteria

A job as part of a grid application can theoretically be of any type: batch, standard application, parallels application, and/or interactive. In the next paragraphs, we will discuss these types.

Batch job

A job in a grid environment could be a traditional batch job on a mainframe or a program invoked via a command line interface in a Windows, Unix, or Linux environment. Normally, arguments are passed to the program, which can represent the data to process and parameter settings related to the job's execution.

Depending on its size and the network capacities, a batch job can be sent to the node along with its arguments and remotely launched for execution. The job can be a script for execution in a defined environment (for example, REXX, Java, or

Perl script), or an executable program that has few or no special requirements for operating system versions, special DLLs to be linked to, JAR files that need to be in place or any other special environmental conditions.

The client, portal, and/or broker may need to know the specific requirements for the job so that the appropriate resource can be allocated.

The data for its computation is either transmitted as arguments or accessible by the job, be it in local or remote storage or in a file that can also be sent across the grid.

A batch job, especially one with few environmental requirements, is generally well suited for deployment in a grid environment.

Standard application

A grid environment can also be applicable to a standard application, like spreadsheets or video rendering systems. For example, if extensive financial calculations on many variations of similar input parameters are to be done, these could be processed on one or more nodes within the grid.

Often, such a standard application requires an installation procedure and cannot be sent over the network to run simply as a batch job. However, a provided command line interface can be remotely used on a grid for execution of the application where it is installed.

In this case, the grid broker or grid portal needs to know the location of the application and the availability of the node. The location of the applications on the grid is relatively fixed, meaning that in order to change it, a new installation has to be performed and the application may need to be registered with the grid portal or grid server before it can be used.

New installations are mostly done manually since the applications often require certain OS conditions and application settings, or very often when installing on Windows, a reboot needs to be executed. This makes standard applications in many cases quite difficult to handle on a grid, but does not exclude them. As advances in autonomic computing provide for self-provisioning, there will be less restrictions in this area.

Using standard software as jobs within a grid could raise licensing issues, either due to the desire to have the application installed on many different nodes in the grid, or related to single-user versus multi-user license agreements. For a more detailed discussion of licenses in a grid environment, refer to redbook *Enabling Applications for Grid Computing with Globus*, SG24-6936-00.

Parallel applications

Applications that already have a parallel application flow, such as those that have been designed to run in a cluster environment, may already be suited to run in a grid environment. In order to allow a grid server or grid portal to make the most advantage of these, there need to be identifiable and accessible handles to the inner functions/jobs of such a parallel application. If this is not the case, such an application can only be handled as one unit, similar to a standard application.

However, it makes sense to include such an application in a grid if the overall task requires more than the resources available in a given cluster. This means that the grid could include several clusters with copies of a parallel application.

Interactive jobs

Interaction with a grid application is most commonly done via the grid portal or grid server interface. This implies that except for when launching the job, there should not be ongoing interaction between the user and the job.

Of course, if we go back to our initial view of the grid as a virtual computing resource, it is certainly plausible to think of an application requiring user interaction being launched on any appropriate resource within the grid as long as a secure and reliable communications channel could be created and maintained between the user and the resource.

Although the GSI-Enabled SSH package is available and could be used to create a secure session, the Globus Toolkit does not provide any tools or guidance for supporting such an application.

6.5.3 Programming language considerations

Whenever an application is being developed, the question of the programming language to be used arises. The grid environment may include additional considerations.

Jobs that are made for high-performance computing are normally written in languages such as C or Fortran. Those jobs whose individual execution time does not play the most important role for the application, but whose contents and tasks are of more importance, may be written in other languages such as Java, or in scripting languages such as Perl.

Within a single grid application, you might even consider writing various parts in different languages, depending on the requirements for the individual jobs and available resources. Some of the key considerations include:

Portability to a variety of platforms

This includes binary compatibility where languages such as Java provide an advantage, since a single binary can be executed on any platform supporting the Java Virtual Machine. Interpreted languages such as Perl also tend to be portable, allowing the application to run no matter what the target platform.

Portability of source code can also be considered. For instance, you may decide to develop an application using C, and then compile it multiple times for a variety of target platforms. This will require additional work by the infrastructure to ensure that appropriate executables are distributed to any target resource.

Runtime libraries/modules

Depending on the language and how the program is linked, there may be a requirement for runtime libraries or other modules to be available. Again, the successful running of an application will depend on these libraries being available on, or moved to, the target resource.

Interfaces to the grid infrastructure

If the job must interface with the grid infrastructure, such as the Globus Toolkit, then the choice of language will depend on available bindings. For example, Globus Toolkit 2.2 includes bindings for C. However, through the CoG initiative, there are also APIs and bindings for Java, Perl and other languages. Note that an application may not have to interface with the Globus Toolkit directly, since it is more the responsibility of the infrastructure that comes into play. That is, given an appropriate infrastructure, the application may be developed such that it is independent of the grid-specific services.

One of the driving factors behind the OGSA initiative is to standardize the way that various services and components of the grid infrastructure interface with one another. This provides programming language transparency between two communicating programs. That is, a program written in C, for example, could communicate with or through a service that is written in another language.

6.5.4 Job dependencies on the system environment

As shown earlier, a grid application does not require a homogenous runtime environment, but there are certain considerations to be taken into account in order to plan for the most beneficial deployment.

For any job in a grid application, the following environmental factors may affect operation. When developing an application, one must consider these factors and either design it to be as independent of these factors as possible, or understand that any dependencies will need to be taken into account within the grid infrastructure.

- Important considerations are the operating system version, service level, and OS parameter settings that are necessary for execution of the job, as well as reliance on certain system services and auxiliary programs such as a registry. It is worthwhile to consider whether the grid application will be capable of running its jobs on any node with different operating systems or whether it will be restricted to a single operating system.
- The memory size required by a job may limit the possible nodes on which it can run. The available memory size depends not only on its physical presence at a node, but also on how much the operating system is capable of granting at runtime.
- DLLs that are to be linked for the execution of the job either need to be available on the target resource or could possibly be transferred and made available on the resource before the job is executed.
- Compiler settings play a role as compiler flags and locations may be different. For example, subtle differences like bit ordering and number of bytes used for real and integer numbers may cause failures when a job is compiled on a different node or operating system than the one where it will eventually be executed.
- There must be a runtime environment in place and ready to receive the job for execution. For instance, the right JDK or interpreter versions may have to be planned and in place.
- Application Server version and standard as well as its capacity may need to be considered as well as access requirements and services to be used.
- Other applications that are needed to properly run a job have to be in place prior to deployment of the grid application. These applications can be compilers, databases, system services such as the registry under Windows, and so on.
- Hardware devices may be required for certain jobs to perform their tasks. For example, requirements for storage, measurement devices, and other peripherals must be considered when building the application and planning the grid architecture.

When developing the grid application, these prerequisites need to be checked in order to avoid too many restrictions for job execution. A large number of restrictions could mean more complicated enablement as well as limiting the number of possible nodes on which the job will be able to run. Therefore, it is better to restrict such requirements during development of the application such that jobs can run in as generic an environment as possible.

6.5.5 Checkpoint and restart capability

A job within a grid application may be designed to be launched, perform its tasks, and report back to the user or grid portal regarding its success or failure. In the latter case, the same job may be launched for a second time, if it has not changed any persistent data prior to its error state. This process can then be repeated until final successful completion. However, it may make sense that failures be handled by the grid server to allow a more sophisticated way to achieve job completion.

By building checkpoint and restart capabilities into the job and making its state available to other services within the grid, the job could be restarted where it failed, even on a different node.

6.5.6 Job topology

For a grid application, there are various topology-related considerations. There are certain architectural requirements covering the topology of jobs and data.

When designing the grid application architecture, some of the key items to consider are:

- Where grid jobs have to or can run
- How to distribute and deploy them over a network
- How to package them with essential data
- Where to store the executables within the network
- How to determine a suitable node for executing the individual jobs

The following are some factors that should be included when considering the above items:

- Location of the data and its access conditions for the job
- Amount of data to be processed by the jobs
- Interfaces needed for any interaction with certain devices
- Inter-process communication needed for the job to complete its tasks
- Availability and performance values of the individual nodes at time of execution
- Size of the job's executable and its ability to be moved across the network

When developing grid-enabled applications, you may not know anything about the topology of the grid on which they will run. However, especially in the case of an intra-grid that may be put in place to support a specific set of applications, this information may be available to you. In such a case, you may want to structure your application and grid in such a way as to optimize the environment by

considering the location of the resources, the data, and the set of nodes on which a particular application might run.

6.5.7 Passing of data input/output

As defined earlier, any job in the grid application needs to pass data in and out in the way of a data producer and a data consumer.

There are various ways to realize the passing of data input and output that are to be considered during application architecture and design:

- *Command line interface* (CLI) can be a natural way for batch jobs and standard applications to receive data. In this case, the data input normally will not be complex in nature, but consists of certain arguments used as parameters to control the internal flow of the job. Such CLIs can easily be integrated in scripts executed at the system level or within a given interpreter. The transfer of data to the job as a consumer happens immediately at launch time. The amount of data will normally be small. For larger amounts of data, there can be arguments that specify the name of a data file or other data source.
- *Data store* of any kind, such as data files in the file system (local or on a LAN or WAN) or records in a database, a data warehouse or other storage system that is available. These data stores can be used for input as well as output of data given that the required access rights are granted to the job. The transfer of data in can be done anytime before the job executes, and likewise the output data could be read anytime after the job completes, thereby providing flexibility for data movement operations.
- *Message queues*, like those provided by WebSphere MQSeries®, are well suited to be used for asynchronous tasks within a grid application, especially when guaranteed delivery of the data provided to the job and generated by the job is of high importance. A job can access the data queues in various ways, normally using specific APIs for putting or getting data as well as for polling the queue for data waiting for processing. In an environment where message queuing servers are already installed, this type of data passing may be desirable.
- *System return value* is a corresponding case to the CLI and normally a way a batch job or any CLI invoked program will return data, or at least status information about how the job ended. This indicates to the grid server or grid portal the status of the individual job and requires appropriate management. The resulting data of the job may be passed to a data store or message queue for further processing or presentation.
- *Other APIs*; when communicating with Web services, Web servers, application servers, news tickers, measurement devices, or any other external systems, the appropriate conditions for data passing in and out have

to be taken into consideration. In these cases, you may use HTTP, HTML, XML, SOAP, or other high-level protocols or APIs.

As indicated, for a grid application there may not be only one way to pass data for a job, but you may use any combinations of the described mechanisms. It is recommended to program grid jobs in such a way that the data sources and sinks are generically handled for more flexible grid topologies. The optimal solution depends on the environment and the requirements to be considered at the architecture and design phase of the grid application.

6.5.8 Transactions

Handling of transactions in their strict definition of commit and roll-back is not yet well suited for a common grid application. The OGSI does not cover these services. However, a grid application may include subsystems or launch transaction-aware operations to subsystems such as CICS®.

For example, you can use a grid node on Linux on zSeries® to get access to a mainframe operating system like z/OS. The zSeries has a built-in high-speed network called hyper sockets. The operating system just sees a network adapter and you can connect to the data bases and transactions systems running on z/OS.

For information about Linux for zSeries applications, please visit:

http://www.ibm.com/servers/eserver/zseries/solutions/s390da/linuxisv.html

The handling of transactions within a grid application easily becomes quite complex with the given definitions, and it needs to be carefully applied. The added benefits of a grid application may be outweighed by the complexity while implementing transactions.

The future development of the OGSA standard may include transaction handling as a service, though at the moment there is no support.

6.5.9 Data criteria

Any application, at its core, is processing data. This means that we must take a closer look at data being used for and within a grid application.

Data influences many aspects of application design and deployment and determines whether a planned grid application can provide the expected benefits over any other data solution.

A detailed discussion is provided in Chapter 4 of the redbook *Enabling Applications for Grid Computing with Globus*, SG24-6936-00.

6.5.10 Usability criteria

While much of a grid computing solution is involved with infrastructure and middleware, it is still appropriate to consider aspects of the solution that relate to usability.

Traditional usability requirements

Traditional usability requirements address features that facilitate ease-of-use with the system. These features address interaction, display, and affective attributes that provide users with an effective, responsive, and satisfactory means to use the system. Hence, these features must be also be addressed when developing a grid computing solution; in other words, this is "business as usual" and continues to play an important part in establishing the requirements for a grid solution.

Usability requirements are used to:

- Provide baseline guidance to the user interface developers on user interface design.
- Establish performance standards for usability evaluations.
- Define test scenarios for usability test plans and usability testing.

Some of the typical usability requirements established for an IT solution play a role and include:

- Tailorability: what requirements exist for the user to customize the interface and its components to allow optimization based on work style, personal preferences, experience level, locale, and national language?
- Efficiency: how will the application minimize task steps, simplify operations, and allow end-user tasks to be completed quickly?

Usability requirements for grid solutions

Grid solutions must address usability requirements recognizing a variety of user categories that may include:

- End users wishing to log in to the grid, submit applications to the grid, query status, and view results
- Owners/users of donor machines
- Administrators and operators of the grid

Consequently, the typical steps followed to identify these requirements for any solution should continue to be followed when creating a grid solution. In addition, the following items may influence the design of grid solutions.

6.5.11 Installation

The grid solution should provide easy, automatic installation by a non-technical person rather than a systems programmer with the need to modify scripts, recompile software, and so on. The install process should be equally straightforward for host, management, and client nodes regardless of the potentially heterogeneous nature of the nodes in terms of operating system or configuration.

6.5.12 Unobtrusive criteria

Transparency and ease of use, as well as job submission and control, are not obvious items, but are essential for a good grid design. The following should be considered regarding those items:

- The use of a grid should be transparent to the user. The grid portal should isolate the user from the need to understand the makeup of the grid.
- Is documentation available or required for all categories of user including executive level summaries on the nature and use of the grid, programmer and administrative support staff? Where possible, the documentation should provide demos and examples for use.
- Ease of resource enrollment after any installation steps should provide a simple configuration of grid parameters to enable the node and its resources to be a participant on the grid. The administrator of the grid or user of a donor machine should not require special privileges to enroll.
- Ease of job submission should alleviate the need for the user to understand the makeup of the grid, search for available resources, or to have to provide complex parameters other than from the business nature of the application. It may be appropriate to provide multiple channels for job submission including a command line (although this has not typically provided ease of use) and a graphical user interface via the grid portal.

  If the architectures of the grid resources are heterogeneous in nature, the solution should provide automation to hide these complexities and provide tools for compiling applications for multiple execution environments. This could also be considered under portability requirements typically addressed under the non-functional requirements.
- Ease of user and host access control should be provided from a single source with appropriate security mechanisms.

6.5.13 Informative and predictable aspects

The status of the grid must be readily available to continually show the status and operation of the grid. This may include indicators showing grid load or utilization,

number of jobs running, number of jobs queued but not yet dispatched, status of hosts, available resources, reserved resources, and perhaps highlighting bottlenecks or trouble spots.

Since the makeup of the grid may be changing dynamically, predicting response times becomes harder. The appropriate trade-offs should be discussed to establish acceptable requirements with associated costs based on the needs of the business.

6.5.14 Resilience and reliability

Some aspects for resilience and reliability of the grid application have already been covered. In this section, they are highlighted from the grid user perspective.

- Particular attention must be paid to the requirements for handling failures. Failures should be handled gracefully. The nature of the application must be understood to identify the correct handling of failures and to provide automatic recovery/restart where possible. Appropriate user notification should be included, recognizing that the actual user may not always be connected to the grid. Consequently, asynchronous mechanisms for feedback might need to be incorporated.
- The nature of applications that are suitable to run on the grid may provide a level of tolerance to failure not typically found in traditional applications. An example of this maybe in the "scavenging" scenario where the application as a whole may be able to tolerate failure of one or more sub-jobs. Since jobs are run on donor machines, the application is subject to the availability of these machines, which are typically outside the application's scope of control. Consequently, the application must tolerate not receiving results from jobs dispatched to these donor machines.
- Applications must be fully integrated with systems management tools to report status and failures. In addition, requirements should be established for how this information will be made available to the end user, indicating the status of their jobs.
- Consideration may also be given to providing intermediate results to an end user when valid results can be achieved.

6.6 Implement the design

At this point, we start the writing process itself.

6.6.1 Write the interface

Accordingly, you should write the service interface in one of two methods allowed by GT3: a Java Interface class or a Grid Web Services Description Language (GWSDL) file. If the grid service is a list of public methods, then you can likely get away with using a Java interface. If your grid service is more complex and will include Service Data, then you are forced to use a GWSDL file.

The service interface includes

- All public methods
- Method parameters
- Method return types
- Service Data Element definitions

It is the nature of the syntax of the Java language that the additional information required for service data cannot be put into a Java interface class. There is a GT3 sub-project called Guide, which contains specialized **`ant`** scripts and tools that look for certain tokens in the Java source file's comments to define service data, but this is not a "mainstream" feature of GT3.

In the future, the GT3 project may choose to take advantage of a new feature going into Java 1.5 called Metadata or Annotations. For details, see `http://www.jcp.org/en/jsr/detail?id=175`. Annotations are extra tokens you can put into a Java source file, but they are actually processed and placed into the .class file by the Java compiler. Other tools will be able to read these annotations in the.class files and perform actions based on what is found. Defining service data and other GT3 features in the Java source file would be a very powerful use of this feature, but it would involve the GT3 community accepting the idea, defining GT3 tokens, writing the .class file processing tools and forcing the GT3 community to use Java 1.5 at a minimum.

6.6.2 Write the implementation

With the service interface fully defined, all that remains is to implement the Java code that performs the requested function when the service's methods are called. This can be as simple or as complex as is required for the application.

The developers leverage the relevant parts of the GT3 toolkit as they develop their code, but this usually represents a small amount of the code above and beyond the code required to implement the overall business solution.

6.6.3 Write the non-Java parts

Due to the nature of GT3 grid services being layered on top of a Web services base, GT3 services must have some non-Java components. These fall into the

following categories, which will be described in much more detail later in the redbook:

- **Data type descriptors** (optional)

  If your service uses a complex data type (a new data type comprised of one or more complex or simple data types), you must define it. The description of the data type is done in an XML syntax and is given the filename extension .xsd.

- **Service interface descriptor**

  This is the file that defines the grid service, very similar to the way a Web Services Description Language (WSDL) file defines a Web service. Since WSDL files do not have a few extra features which are required for the complete definition of a grid service, the grid WSDL (GSWDL) file format was invented. The supplied GT3 tools know how to process GWSDL files and in fact, the next version of the WSDL specification (1.2) will incorporate the features in GWSDL and GT3 will migrate to WSDL.

- **Service deployment descriptor**

  This is the file which provides the deployment instructions to the grid service container, and is similar to the style used for Web services.

6.6.4 Write the clients

With the entire grid service implemented, you must now complete the last step of writing the clients that access the grid service. As described above, the designer must follow the architecture and detailed design on such client-specific items as service life cycle, security, service data/notification, state management, error handling and data formats/external connections.

7

Case study: grid application enablement

This chapter presents a case study which aims to apply the main concepts of grid service application design, specifying and coding over a real practical grid application. The study project is a bulletin service application, as could be implemented by a news organization's Web site.

The main goal of this chapter is to programmatically explain the various features of the grid services that have been specified in OGSA and OGSI and further put into effect in the GT3 reference implementation.

7.1 Introduction

This case study applies the main concepts of grid service application design, specifying and coding to a real practical grid application. The target application is not as complex as a large grid application can be, so some steps of the design methodology have been skipped. Moreover, the process of coding conversions will not be detailed, since it is a repetitive task and has already been shown previously.

The design requirements of this study will use previous knowledge presented in this document. Each bulletin is a short text message representing a newsworthy item. The core of the project is the bulletin service itself and several client applications that drive the flow of bulletins through the service.

Moreover, this study introduces the various steps needed in building a grid application. Hence, this chapter will not focus on the steps and commands needed to compile, test, and deploy the grid application.

7.2 Case study: design

This first part focuses on the identification of requirements to produce a general architecture of the system.

We define high-level functional requirements and non-functional requirements. Note that no particular grid or Globus feature implementation is implied, leaving those decisions to the architects and designers in the following phases of development.

7.2.1 Functional requirements

In this section, we will detail the functionality to be provided by the News Service application (also known as the *bulletin service*). In subsequent chapters, these requirements will be elaborated upon and the design and implementation of the application will be discussed. The main goal of this chapter is to illustrate the various features in GT3. Hence, the application is chosen such that its design covers the different features of GT3. We will first present the functional requirements of the application in the form of a problem statement and expand them with the aid of a system context and use cases.

Problem statement

The underlying application is to be used by News Servicervice organization whose purpose is to electronically publish news bulletin messages to various subscribers who subscribe to the News Service. The News Service organization

publishes bulletin messages within various categories, such as Business News, Sports, and Weather. The organization has various employees playing specific roles; this allows the organization to meet its objective of providing an accurate news bulletin to its subscribers. Particularly important to this application is the fact that it has employees playing the writer, editor, and administrator roles. The writers gather news and submit the news bulletins for approval via this application. The editors are informed of any pending bulletins that the writers have submitted. The editors log on to the application, are authenticated by the application and retrieve the pending news bulletins. Upon review of the news bulletins, they either approve or disapprove of the news bulletins submitted by the writers. All approved news bulletins are subsequently published by the application to all registered subscribers. The administrator is responsible for starting and stopping the application and performing other necessary administrative functions. In addition to the writers in the organization, the News Service organization allows other business partner organizations to submit news bulletins. Upon receipt of news bulletins from the business partner organizations, the administrator loads the news bulletins into the application for further review by the editor and publishing to the subscribers.

System context

In this section, we will illustrate the system context of the problem described previously. The system context represents the entire system or application as a single object or process and identifies the events that are passed between the system and the entities with which it interacts. In turn, it aids in discovering all the use cases that the system or application must implement. Figure 7-1 on page 118 shows the system context for the application to be developed for the problem statement specified in the earlier sub-section.

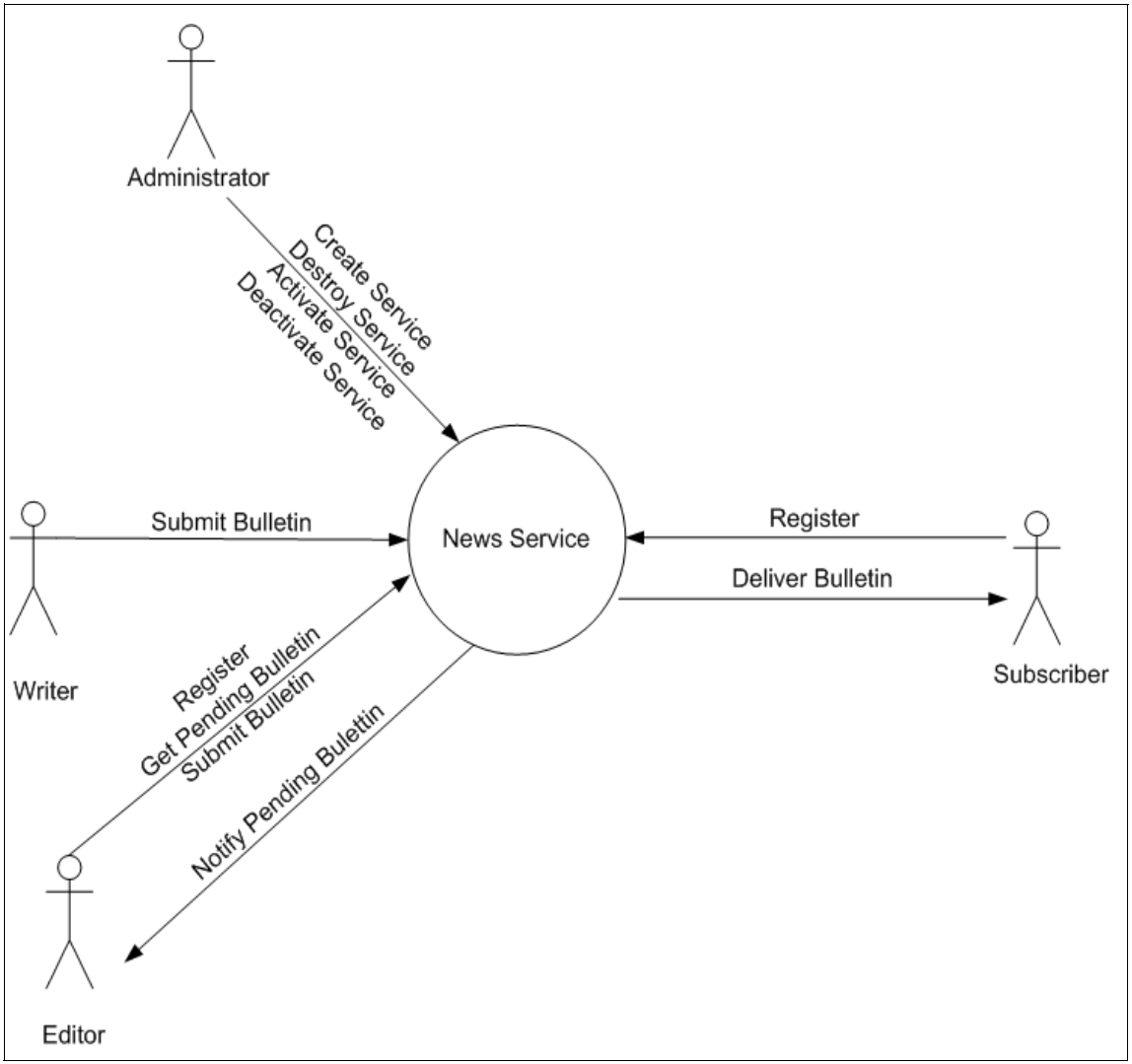


Figure 7-1 The system context

As shown in Figure 7-1 above, the system or application being built is represented by the circle in the center. The various entities or actors it interacts with are represented using stick figures. The events that trigger the application are shown alongside the arrows from the actor(s) toward the system. Similarly, the events that are used to notify the actors are shown in the figure alongside the arrows that point away from the system toward the actors. The administrator, writer and editor are the roles of the employees within the news agency organization. The administrator ensures that the system is up and running, the

writer creates and submits the news bulletin to the News Service and the editor is responsible for approving the bulletin submissions from the writer. The subscriber is the customer of the News Service. He/she registers for the news bulletins and has them delivered by the system. The business partner also contributes to the news agency and submits the bulletin. In the next section, we describe the roles of the various actors in detail and also describe the use cases derived from the system context.

Use case model

In this section, we will present the use case model of the solution. A use case model formalizes the functional requirements of the application under development. The model uses graphical symbols and text to specify how users in specific roles will use the system (use cases). The textual descriptions of use cases are from a user's point of view; they do not describe how the system works internally or explain its internal structure or mechanisms. The use cases, in addition to formalizing the requirements, provide an initial structure for the application to be implemented.

The use case model is shown in Figure 7-2 on page 120. The stick figures on the left show the various actors using the application. The actors represent the users in specific roles that use the application. The ovals are the use cases that the application supports. A use case represent an individual functional behavior of the application that is triggered by an actor’s action. The arrow from the actor to the use case shows the communication association between the actors and the use cases. The application will implement the specified use cases. In subsequent sections, we will illustrate the various components that are needed for implementing the use cases of the application. The application will be implemented using grid services and hence the components will be designed based on a grid services implementation. In addition, we will specify the design of the various classes within the components and the interaction diagrams for implementing the various use cases of the system.

The actors and the associated use cases they trigger are as follows.

Administrator

The administrator is the role within the news organization that manages the application. Specifically, he/she will perform or trigger four use cases, as shown in Figure 7-2 on page 120. He/she will use the user interface provided by the application and trigger the **`Create Service`** use case to create a service for a specific topic, such as for Sports. He/she will trigger the **`Destroy Service`** use case to destroy the service. In addition, he/she can further activate and de-activate the service by triggering the **`Activate Service`** and **`Deactivate Service`**, respectively.

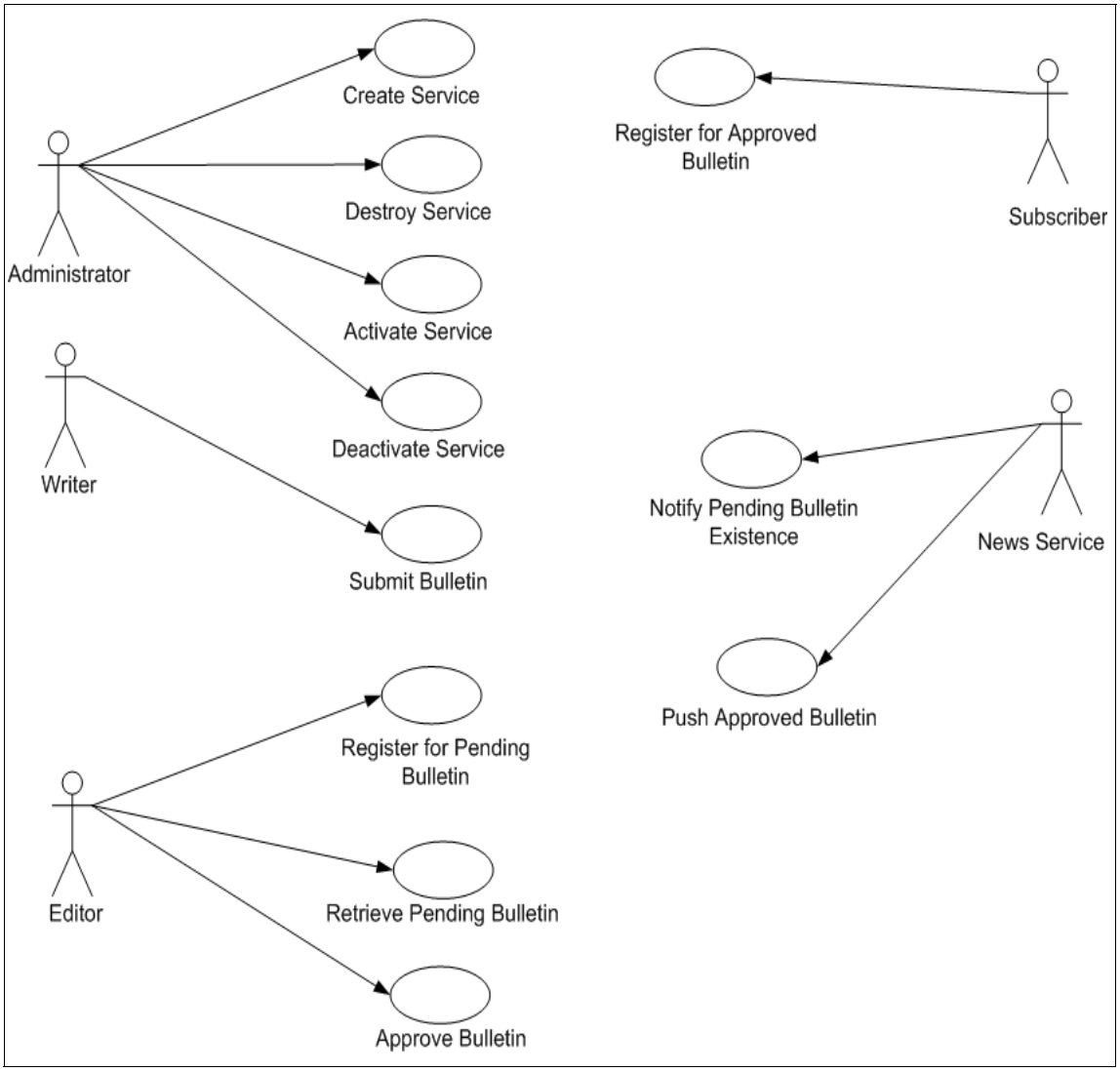


Figure 7-2 Use case model

Writer

The writer is mainly responsible for submitting the news messages to the News Service. He/she triggers the **`Submit Bulletin`** use case. The writer will research and/or investigate news articles to be published and submit news bulletins for the various news categories for which the News Service decides to provide News Service, such as Sports, Weather, and Business News.

Editor

The editor is an approving authority in the News Service organization. He/she triggers three use cases, namely, **`Register for Pending Bulletin`**, **`Retrieve Pending Bulletin`** and **`Approve Bulletin`**. In order to perform this role, the editor needs to be made aware of the news bulletins submitted by the writers via the **`Submit Bulletin`** use case. The application needs to be aware of the specifics of the editor in order to inform the editor of the pending bulletin messages. The editor triggers the **`Register for Pending Bulletin`** use case in order to register with the application for the receipt of pending bulletins. Upon registration, the application registers the editor's reference and notifies the editor whenever the application has news bulletins submitted by the writer. During the execution of **`Retrieve Pending Bulletin`**, the editor retrieves the news bulletins and reviews them. Upon satisfactory review of the news bulletins, the editor executes the **`Approve Bulletin`** use case. The **`Approve Bulletin`** use case moves the approved news bulletins from the pending queue to the approved queue. All approved messages can be viewed by the subscribers of the News Service. It should be noted that the editor will execute **`Retrieve Pending Bulletin`** and **`Approve Bulletin`** only after he/she is notified by the News Service server of the existence of pending news bulletins. As stated in the description for the News Service, it executes the **`Notify Pending Bulletin Existence`** use case to notify the writer.

Subscriber

The subscriber is the customer of the news agency. The subscriber receives news bulletins from a specific News Service instance, such as the sports News Service instance. For this, the subscriber needs to register with the news agency, by triggering the **`Register for Approved Bulletin`** use case.

The service delivers all the news bulletins for the service registered for by the subscriber when the news bulletins are approved by the editor. As described below, the News Service executes the **`Push Approved Bulletin`** to notify the subscriber client as to when the editor client approves the news bulletins.

News Service

The News Service is the heart of the application and enables the workflow between the different actors. The News Serice core workflow has to notify the editor and the subscribers at the appropriate time. It notifies the editor whenever the writer submits a news bulletin by triggering the **`Notify Pending Bulletin Existence`** use case. The News Service is aware of the editor because the editor has registered with the News Service using the **`Register for Pending Bulletin`** use case. Similarly, the New Service notifies the subscriber whenever the editor approves the news bulletin using the **`Push Approved Bulletin`** use case. As stated above, the subscriber registers for this notification using the **`Register for Approved Bulletin`**. The key difference between the two use cases executed by

the News Service is that the **`Notify Pending Bulletin Existence`** use case is a pull notification and the **`Push Approved Bulletin`** use case is a push notification. Using a pull notification, no bulletin is transferred to the editor during notification; rather, the editor is notified of the existence of the pending news bulletins. The editor subsequently has to retrieve the pending bulletins. On the other hand, using a push notification, the News Service transmits the approved bulletin to the subscriber along with the notification. The subscriber does not have to subsequently retrieve the news bulletin.

7.2.2 Non-functional requirements

Non-functional requirements of an IT system or an application are quality requirements or constraints of the system that must be satisfied. Unlike functional requirements, which focus on the desired functionality of the system, non-functional requirements address major operational areas of the system or application and are specified in order to ensure the robustness of the system or application. The analysis of non-functional requirements must be considered in a real application, as shown in 6.3.2, “Non-functional requirements” on page 78. For our application example, only the robustness and reliability of the bulletin services are being considered.

7.2.3 Architecture overview

In this section, we will provide an overview of the architecture that was designed to implement the requirements specified in earlier sections. We will explain the architecture using an overview diagram. In addition, we will discuss the architectural decisions which were taken. Figure 7-3 on page 123 shows an overview of the application’s architecture.

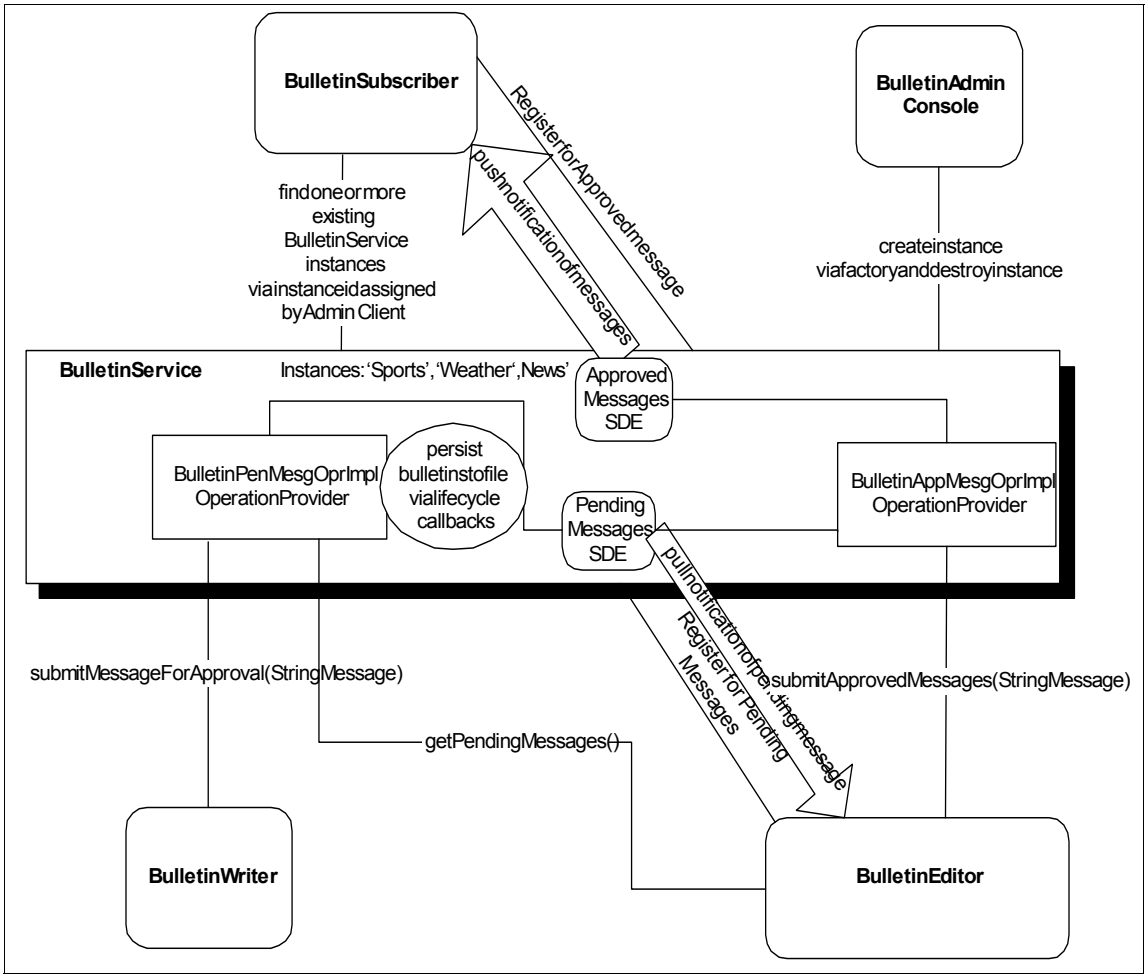


Figure 7-3 Architecture overview

The following are the key architectural decisions:

1. The requirements are implemented as a distributed application using the grid service. The key functionality of the application is encapsulated in a single grid service. As shown in Figure 7-3, the shaded box in the center represents the implementation of the single grid service. The various use cases specified earlier for the administrator, writer, editor, and subscriber are implemented by the single grid service. As will be described later, the service implementation is deployed in a GT3 stand-alone container.
2. The user interface for the administrator is implemented by the administrator client application. As shown in Figure 7-3 on page 123, the administrator

client application communicates with the grid service server to implement the use cases triggered by the administrator. The link between the administrator client and the bulletin service depicts the communication between the client and the server.

3. Similarly, the user interfaces for the writer, editor and subscriber are implemented by the writer client, editor client, and subscriber client, respectively. Just as in the case of the administrator client, the user interfaces implement the triggering and execution of the appropriate actors and communicate with the grid service implementation in providing the functionality.
4. The administrator client will use the Factory of the bulletin service in creating different instances of the bulletin service. One service instance will be created for each bulletin topic. Hence, for three topics of Sports, Weather, and Business News, three separate instances of the grid service will be created.
5. The grid service will utilize two Service Data Elements (SDEs) to store the *pending* bulletins that are submitted by the writer and the *approved* bulletins that are approved by the editor, respectively. The SDEs are shown in Figure 7-3 on page 123 as two rounded rectangles inside the bulletin service.
6. The editor client and the subscriber will implement the **`Register for Pending Bulletin`** and the **`Register for Approved Bulletin`** use cases, respectively, by using the GT3 runtime. As shown in Figure 7-3 on page 123, the editor will register for notification of the existence of new pending messages. The subscriber will register for changes to the approved bulletin SDE.
7. The **`Notify Pending Bulletin Existence`** use case is triggered by the grid service and is implemented by the editor client. The arrow from the service to the editor client represents the communication in the implementation of the use case. Similarly, the **`Push Approved Bulletin`** use case is triggered by the grid service and implemented by the subscriber client (or bulletin client). The arrow from the service to the bulletin client illustrates the push communication.
8. The service operations to support the use cases for the writer and the editor will be implemented in two separate operation providers, namely the **`Submit Bulletin Operation Provider`** and the **`Approve Bulletin Operation Provider`**, respectively. The operation providers are shown as two rectangles inside the billeting service. The **`Submit Bulletin Operation Provider`** use case will provide the implementation for submit message and the **`Approve Bulletin Operation Provider`** use case will provide implementation for the get pending bulletin and the approve bulletin.
9. In order to make the grid service robust, the pending bulletins will be persisted to a non-volatile file system when the grid service is terminated. Similarly, when the service instance is instantiated at the time of server startup, the bulletins will be retrieved from the file system and populated in the

SDE. This feature is supported to fulfill the robustness non-functional requirement. The life cycle callback feature of a grid service will be used to implement this functionality. This is shown in Figure 7-3 on page 123 as an oval inside the bulletin service.

The following table (Table 7-1) helps to match the method names, operation names, and use case names.

Table 7-1 Method, operation and use case names

Use case name	Actor	Name of method
Submit Bulletin	writer	submitMessageForApproval()
Retrieve Pending Bulletin	editor	getPendingMessages()
Approve Bulletin	editor	submitApprovedMessages()
Notify Pending Bulletin Existence	service	notifyChange()
Push Approved Bulletin	service	deliverNotification()

7.3 Case study: grid service specifying and coding

This part of the redbook focuses on specifying and coding the application. Hence, this chapter will not focus on the steps and commands needed to compile, test, and deploy the grid application. The main intention of this chapter is to programmatically explain the various features of the grid services that have been specified in the standards OGSA and OGSI, and further implemented in the Globus Toolkit V3.

The application will be progressively built in four stages. At each stage, we will introduce a sub-set of the grid service features and explore its usage and considerations in building the application. Each subsequent stage will be built on the previous phase and enhance it with additional features, and should follow the complete development methodology for each phase.

7.4 Phase I: building the core News Service

In the first phase, we will start by building the core of the News Service application. The essential elements of any grid service are the grid server and the grid client. In our application, there is one grid service: Bulletin Service. The grid server implements the business functionality and the grid client uses the functionality provided by the server. Further, in order to manage the server, we need an administration client. The administrator starts up and shuts down the functionality made available by the server. Hence, we will implement the key use cases for the administrator and the subscriber in addition to implementing the required service server functionality.

Here, we will focus on the support of the two main actors specified in the previous chapter, namely, the subscriber and the administrator. To support the administrator, we will implement the create service and the destroy service use case. We will support the subscriber by combining the two use cases (register for approved bulletin and push approved bulletin) into a single use case (retrieve bulletin). The retrieve bulletin use case will allow the subscriber to obtain a set of messages by accessing the service.

We will now summarize the decisions specified in 7.2.3, “Architecture overview” on page 122’ and which are relevant to this phase of the development. In this phase, we will accomplish the following:

1. Encapsulate the core functionality in a single grid service, namely, the bulletin service.
2. Implement the administrator user interface as a single administrator client module.
3. Implement the subscriber user interface as a single subscriber client module.
4. Allow the administrator to create and destroy three instances of the service for each of the three topics the news agency supports, namely, Sports, Weather, and Business News.
5. Implement the application using the Java language and the GT3-provided tools and runtime.

Support of the aforementioned functionality requires the use of the following features of grid services:

1. The generation and development of the skeleton code comprised of the server side service implementation, client side service implementation, server side stub, and client side stub.
2. The use of the *Factory* interface of the grid service in the creation of the grid service instances.

We will emphasize these grid service features when we illustrate the implementation of the core News Service.

7.4.1 Development of server-side functionality

This section discusses the implementation of the server service, the bulletin service.

Bulletin service definition

As specified earlier, the first step in building the bulletin service is the specification of the service. We will explain the GWSDL in sections. Figure 7-4 on page 127 shows the first section of the specification of the bulletin service in GWSDL. As shown, the name of the service, BulletinService, is specified first along with the various required namespaces. The first namespace defines the target namespace for the BulletinService. Types specific to the bulletin service are stored in that namespace. The namespaces with `OGSI`, `gwsdl`, and `sd` tags are specific to the various types defined for the grid services and must be included. Similarly, the namespace with the `xsd` tag defines the XML schema of the various elements used for the definition of Web services. The last namespace specifies the namespace for the definition of SOAP messages. Since SOAP bindings are being used, that namespace is required. Please refer to Chapter 2, “Service Oriented Architecture” on page 5 for a detailed description of the various elements within a WSDL file and in SOAP messages.

```
<?xml version="1.0" encoding="UTF-8"?>
<definitions name="BulletinService"
    targetNamespace="http://bulletin.gt3.grid.itso.ibm.com/common/stubs"
    xmlns:tns="http://bulletin.gt3.grid.itso.ibm.com/common/stubs"
    xmlns:ogsi="http://www.gridforum.org/namespaces/2003/03/OGSI"
    xmlns:gwsdl="http://www.gridforum.org/namespaces/2003/03/gridWSDLExtensions"
    xmlns:sd="http://www.gridforum.org/namespaces/2003/03/serviceData"
    xmlns:xsd="http://www.w3.org/2001/XMLSchema"
    xmlns="http://schemas.xmlsoap.org/wsdl/">

<import location="../../ogsi/ogsi.gwsdl"
namespace="http://www.gridforum.org/namespaces/2003/03/OGSI"/>
```

Figure 7-4 Bulletin service specification in GWSDL - part 1

Subsequent to the definition of the service name and namespaces, the data type definitions used by the messages exchanged between the service requestor and the service provider are specified in the `types` section. Figure 7-5 on page 128 shows the type definitions for our service.

```
<types>
<xsd:schema targetNamespace="http://www.itso.ibm.com/namespaces/grid/Bulletin"
        attributeFormDefault="qualified"
        elementFormDefault="qualified"
   xmlns="http://www.w3.org/2001/XMLSchema">
  <xsd:element name="submitMessageForApproval">
    <xsd:complexType>
      <xsd:sequence>
        <xsd:element name="value" type="xsd:string"/>
      </xsd:sequence>
    </xsd:complexType>
  </xsd:element>
  <xsd:element name="submitMessageForApprovalResponse">
    <xsd:complexType/>
  </xsd:element>
  <xsd:element name="getPendingMessages">
    <xsd:complexType/>
  </xsd:element>
  <xsd:element name="getPendingMessagesResponse">
    <xsd:complexType>
<xsd:sequence>
        <xsd:element name="value" type="xsd:string"/>
      </xsd:sequence>
    </xsd:complexType>
  </xsd:element>
  <xsd:element name="submitApprovedMessages">
    <xsd:complexType>
      <xsd:sequence>
        <xsd:element name="value" type="xsd:string"/>
      </xsd:sequence>
    </xsd:complexType>
  </xsd:element>
  <xsd:element name="submitApprovedMessagesResponse">
    <xsd:complexType/>
  </xsd:element>
</xsd:schema>
</types>
```

Figure 7-5 Bulletin service specification in GWSBL - part 2

Further, the various messages that are transferred between the service requestor and service provider are described, including the parameters for the messages. A message represents one interaction between the service requestor and the service provider. Figure 7-6 on page 129 shows the message definition for our service.

```
<message name="SubmitForApprovalInputMessage">
  <part name="parameters" element="tns:submitMessageForApproval"/>
</message>
<message name="SubmitForApprovalOutputMessage">

<part name="parameters" element="tns:submitMessageForApprovalResponse"/>
</message>
<message name="GetPendingInputMessage">
  <part name="parameters" element="tns:getPendingMessages"/>
</message>
<message name="GetPendingOutputMessage">
  <part name="parameters" element="tns:getPendingMessagesResponse"/>
</message>
<message name="SubmitApprovedInputMessage">
  <part name="parameters" element="tns:submitApprovedMessages"/>
</message>
<message name="SubmitApprovedOutputMessage">
  <part name="parameters" element="tns:submitApprovedMessagesResponse"/>
</message>
```

Figure 7-6 Bulletin service specification in GWSBL - part 3

Finally, the port type within the service is defined. A port type is a collection of operations supported by the grid service. Figure 7-7 on page 130 shows the definition of the port type for our service. The operations that are part of a port type are enclosed with the port type section. Our example BulletinService consists of one port type, namely, BulletinPortType.

```
<gwsdl:portType name="BulletinPortType" extends="ogsi:GridService">
  <operation name="submitMessageForApproval">
    <input message="tns:SubmitForApprovalInputMessage"/>
    <output message="tns:SubmitForApprovalOutputMessage"/>
    <fault name="Fault" message="ogsi:FaultMessage"/>
  </operation>
  <operation name="getPendingMessages">
    <input message="tns:GetPendingInputMessage"/>
    <output message="tns:GetPendingOutputMessage"/>
    <fault name="Fault" message="ogsi:FaultMessage"/>
  </operation>
  <operation name="submitApprovedMessages">
    <input message="tns:SubmitApprovedInputMessage"/>
    <output message="tns:SubmitApprovedOutputMessage"/>
    <fault name="Fault" message="ogsi:FaultMessage"/>
  </operation>
</gwsdl:portType>
</definitions>
```

Figure 7-7 Bulletin service specification in GWSBL - part III

It should be noted that various extensions for the grid service are defined in this section of the GWSDL with the `gwsdl`, `ogsi`, and `sd` tags. The namespaces for these extensions are specified in the namespaces section of the specification, as described in the beginning of this section. In order to be a grid service, our port type must inherit from or extend the ogsi:GridService port type. Also, since our service is a source of notifications to the clients of the service, specifically the editor client and the subscriber client, the BulletinPortType also extends the ogsi:NotificationSource interface. It should be noted that the extends attribute of the port type is a grid service extension to Web services, and hence the extension in GWSDL from the WSDL.

Our port type has three operations defined, namely, **`submitMessageForApproval`**, **`getPendingMessages`**, and **`submitApprovedMessages`**. The input, output and fault messages for the operations are also specified. The type of the fault message is a grid service extension. Our port type also has two Service Data Elements, namely PendingMessages, and ApprovedMessages. Service data are also grid service extensions.

As mentioned earlier, the client and server stubs are generated from the service specification file. Please refer to Chapter 4, “Grid services development” on page 37 for details. Since the stubs are transparent to the developer, they will not be shown or discussed here. Interested users are advised to look at the ~/Bulletin/common/ directory for the list of generated classes and interfaces and the details of the code.

Server-side bulletin service implementation

Subsequent to the generation of the client- and server-side stubs, the server-side functionality is implemented.

In order for a class to provide the implementation of one or more operation(s) within the port type, the class must implement the OperationProvider Java interface. The implementation of the OperationProvider interface has to be done in addition to the implementation of the operation(s) of the port type. Figure 7-8 shows the OperationProvider interface. The implementation class must implement the initialize() and getOperation() methods in addition to other implementation methods.

```
package org.globus.ogsa;
import javax.xml.namespace.QName;


public interface OperationProvider {

public void initialize(GridServiceBase serviceBase)
        throws GridServiceException;
public QName[] getOperations();

}
```

Figure 7-8 OperationProvider interface

The initialize() method is called when the operation provider is added to the grid service. The serviceBase argument specifies the service this provider is being associated with. The getOperations() method is called during initialization when the grid service needs to find out what operations are supported by this provider. It must return an array of operation QNames as defined in the GWSDL.

In our example, we implement the operations in the BulletinOprImpl class. The example implementation of the class is shown next. It should be noted that the class provides the implementation of the initialize() and the getOperations() methods. The class stores the reference to the grid service associated with this operation provider when the initialize method is invoked by the runtime during the initialization process. In this phase, we are not implementing any of the grid service operations. In subsequent phases, we will build upon the skeleton code developed in this phase and implement the methods.

```
package com.ibm.itso.grid.gt3.bulletin.server;

import org.globus.ogsa.GridServiceBase;
import org.globus.ogsa.GridServiceException;
import org.globus.ogsa.OperationProvider;

import javax.xml.namespace.QName;

public class BulletinOprImpl implements OperationProvider {
   // Operation provider properties
   private static final QName[] operations = new QName[]{new QName("", "*")};
   private GridServiceBase base;


   public void initialize(GridServiceBase base) throws GridServiceException {
      this.base = base;
   }

   public QName[] getOperations() {
      return operations;
   }

   }
```

Figure 7-9 BulletinOprImpl class implementation

7.4.2 Administration client implementation

In this section, we will implement the administration client. As mentioned before, the admin client is responsible for creating and destroying the instances of the bulletin grid service. Figure 7-10 on page 133 shows the BulletinAdminConsole class, which implements the administration client console. The class takes the GSH of the factory that creates the bulletin service. Please refer to 5.2, “Factory” on page 60 for a description of GSH and GSR.

```
package com.ibm.itso.grid.gt3.bulletin.client;

import org.globus.ogsa.utils.GridServiceFactory;
import org.gridforum.ogsi.Factory;
import org.gridforum.ogsi.LocatorType;
import org.gridforum.ogsi.OGSIServiceGridLocator;

import java.net.URL;
import java.io.*;
import java.util.*;

public class BulletinAdminConsole {
public static void main(String[] args) {
       String addr = new String();
       if(args.length != 1) {
          System.out.println("Usage: java
com.ibm.itso.grid.gt3.bulletin.client.BulletinAdminConsole <GSH>");
          //addr = new
String("http://10.3.5.84:8080/ogsa/services/sample/bulletin/BulletinFactory");
          return;
       } else {
          addr = args[0];
       }

       System.out.println("WELCOME TO BULLETIN ADMIN CONSOLE...");
       System.out.println("Command Help...");
       System.out.println(">c <instance name>  : to create a instance");
       System.out.println(">d <instance name>   : to destroy a instance");
       System.out.println(">q                   : to exit this program");
       System.out.println();
```

Figure 7-10 Implementation of the bulletin administrator console

As shown in Figure 7-10, the code lets the user specify the creation or destruction of the Bulletin Service instances. The user is asked to provide the name of the instance to be created or destroyed along with the option **c** or **d** for creation and deletion, respectively. The user exits the program by entering the **q** option.

Figure 7-11 on page 134 is the continuation of the Bulletin Administrator Console. The console has to obtain the Grid Service Reference (GSR) of the factory before it can create any object. The GSR of the factory is obtained from the GSH of the factory, provided by the user during console invocation. As shown in the code below, the URL of the GSH is first obtained. An instance of the OGSIServiceGridLocator is created. This instance provides the locator service and aids in obtaining the corresponding GSR for a given GSH. We create an

instance of the GridServiceFactory. This factory instance is subsequently used for creating bulletin grid service instances.

```
try{
          //Get command-line argument
          URL GSH = new java.net.URL(addr);

          //Get a reference to the Bulletin Service Factory
          OGSIServiceGridLocator ogsiServiceGridLocator = new OGSIServiceGridLocator();
          Factory factory = ogsiServiceGridLocator.getFactoryPort(GSH);
          GridServiceFactory gridServiceFactory = new GridServiceFactory(factory);

          BufferedReader buffer = new BufferedReader(new InputStreamReader(System.in));
          Hashtable instanceStore = new Hashtable();

          System.out.println("please input command...");
```

Figure 7-11 Implementation of the bulletin administrator console - continued

Figure 7-11 is the continuation of the bulletin administrator console. The key lines of code are highlighted. As can be seen in the figure, if the user enters the option to create the instance and passes the name of the instance to be created, the instance of the bulletin service is created by invoking the createService() method on the grid service factory instance. When an instance is created, the createService() returns the GSH of the created instance. We store the GSH of the created instances in the instanceStore hash table. The created instances are hashed based on the names of the instances.

When the user requests the destruction of the instance, he/she enters the option **d** along with the name of the service instance. The code looks up the GSH of the Bulletin service instance to be deleted from the instanceStore hash table based on the name of the instance. Subsequently, the GSR of the instance to be deleted is obtained by invoking the getGridServicePort() method on the ogsiServiceGridLocator instance. Once the reference to the instance is obtained, it is destroyed by invoking the destroy() method. The Admin console destroys all the instances that were created by it when the user chooses the **q** option.

In summary, the bulletin administrator console code discussed above illustrates the implementation of the following features of grid services:

1. The mechanism to obtain a GSR once a GSH has been provided.
2. The creation of a grid service instance using the factory of the grid service.

3. The creation of multiple instances of the same grid service. This is contrary to the Web service implementation: different instances of the same Web service cannot be created.
4. The mechanism to destroy the grid service instance.

7.4.3 Subscriber client implementation

In this section, we will implement the skeleton of the bulletin subscriber client. Since we have not implemented any of the grid service operations as of yet in this phase, we will not show the invocation of the service operations by the bulletin subscriber client. However, we will focus on the location of the grid service instance by the bulletin subscriber client. Figure 7-12 on page 136 shows the implementation of the bulletin subscriber client.

As shown in Figure 7-12 on page 136, the subscriber client functionality is encapsulated in the run() method. When the subscriber client is invoked, the address of the bulletin service and the name(s) of the service instances to which it is subscribing are passed as arguments. The key functionality is highlighted in bold. The implementation constructs the GSRs of the instance name(s) passed, in URL form, by concatenating each of the instance names with the address of the bulletin service.

```
package com.ibm.itso.grid.gt3.bulletin.client;

import com.ibm.itso.grid.gt3.bulletin.common.stubs.service.BulletinServiceGridLocator;
import com.ibm.itso.grid.gt3.bulletin.common.stubs.BulletinPortType;

import java.net.URL;

public class BulletinSubscriber {

   BulletinPortType[] bulletins;

   public static final void main(String[] args) {
      BulletinSubscriber bs = new BulletinSubscriber();
      bs.run(args);
   }

   private void run(String[] args) {
// The base GSH and the instance names will be informed
      // as command-line arguments
      String baseGSH = args[0];

      // Build service instance references
      int instanceCount = args.length-1;
      BulletinPortType[] bulletins = new BulletinPortType[instanceCount];

      try {
         String sinks[] = new String[instanceCount];

         for (int i=0; i<instanceCount; i++) {
            String GSHstr = baseGSH+args[1+i];
            URL GSH = new URL(GSHstr);
            BulletinServiceGridLocator bulletinGL = new BulletinServiceGridLocator();
            bulletins[i] = bulletinGL.getBulletinServicePort(GSH);
         };

         // Now we can issue any call to the service instances
         // referenced by the port types in the "bulletins[]" array

         System.out.println("Type any key to exit.");
         System.in.read();
      } catch (Exception e) {
         e.printStackTrace();
      }
   }
}
```

Figure 7-12 Bulletin subscriber client Implementaiton

After the creation of the GSH of the bulletin service instances, the references are resolved into GSRs. An instance of the BulletinServiceGridLocator instance class aids in resolving GSHs into GSRs. As shown in the code, the subscriber client creates an instance of the BulletinServiceGridLocator class and invokes the getBulletinServicePort() method on it by passing in the GSH. Obtaining a reference to the bulletin services instances is critical to the support of other subscriber client functionality. In subsequent phases, we will expand the subscriber implementation.

7.5 Phase II: operationalizing the News Service with news writer and subscriber notification of news

In the second phase, we will expand the core News Service implementation of the first phase. As we have described in the previous chapter, the News Service has writers who scout for and submit news to the news organization on various topics. Also, in this phase, the subscriber will be allowed to register for the news bulletins and will be notified of the news bulletins as and when they are written by the writer. Hence, we will implement the use case for supporting the writer and the subscriber in addition to implementing the required service server functionality.

Here, we will focus on supporting the two main actors specified in the previous chapter, namely, the writer and the subscriber. To support the writer, we will implement the **`Submit Bulletin`** use case. We will support the subscriber by supporting the **`Register for Approved Bulletin`** and **`Push Approved Bulletin`** use cases. As we had specified in the architectural decisions in the previous chapter, we will implement the writer client interface as a single writer client module. Also, we will implement the **`submitApprovedMessages`** operation in a separate operation provider class within the server to support the writer. It should be noted that in this phase, there is no separate approval authority and the writer both submits the bulletin and approves the message by invoking the **`submitApprovedMessages`** operation.

Support of the aforementioned functionality requires the use of the following features of grid services:

1. The use of the Service Data Element (SDE) in providing state for the grid service.
2. The implementation of the operation provider class in providing the grid service operation.
3. The implementation of the push notification in supporting the notification of state changes of the SDE.

7.5.1 Enhancing server-side functionality

In this section, we will augment the skeleton server-side code developed in the previous phase. In particular, we will implement the SDE that carries approved message notifications to the clients and the **submitApprovedMessages** operation. In the previous phase, we had introduced the BulletinOprImpl class which implemented the OperationProvider interface. We had also stated that this class will be used to provide the implementation of the operations of the grid service. We illustrate the code for the BulletinOprImpl class for this phase in a set of figures shown next. Additions to the code from the previous phase are highlighted in bold.

Bulletin service definition

The introduction of Service Data Elements implies some important changes in the GWSDL file. The excerpt below was taken from the new version of the GWSDL file and the changes are shown in bold.

```
<?xml version="1.0" encoding="UTF-8"?>
<definitions name="BulletinService"
    targetNamespace="http://bulletin.gt3.grid.itso.ibm.com/common/stubs"
    xmlns:tns="http://bulletin.gt3.grid.itso.ibm.com/common/stubs"
    xmlns:ogsi="http://www.gridforum.org/namespaces/2003/03/OGSI"
    xmlns:gwsdl="http://www.gridforum.org/namespaces/2003/03/gridWSDLExtensions"
    xmlns:sd="http://www.gridforum.org/namespaces/2003/03/serviceData"
    xmlns:data="http://bulletin.gt3.grid.itso.ibm.com/common/stubs"
    xmlns:xsd="http://www.w3.org/2001/XMLSchema"
    xmlns="http://schemas.xmlsoap.org/wsdl/">

<import location="../../ogsi/ogsi.gwsdl"
namespace="http://www.gridforum.org/namespaces/2003/03/OGSI"/>

<import location="MessageDataType.xsd"
namespace="http://bulletin.gt3.grid.itso.ibm.com/common/stubs"/>

<types>
...
```

Figure 7-13 Segment of the revised GWSDL showing changes in namespaces and import statements

As can be seen in Figure 7-13, two namespaces have been added to the attributes section of the definitions tag. The first namespace refers to the set of definitions required for declaring Service Data Elements. As a matter of fact, this namespace must always be declared when Service Data Elements are defined.

The second namespace refers to a data type that has been introduced for storing the service data itself. When defining service data, its data type must be declared

as well. The namespace of this data structure and its binding tag are arbitrary; in this case, we have chosen to define it at the same namespace as the service itself and to bind it to the tag *data*.

The definition of the data structure itself is contained in the file named MessageDataType.xsd, which is referred to in the new (and highlighted) import statement. The contents of this file are presented in Figure 7-14.

```
<?xml version="1.0" encoding="UTF-8"?>
<wsdl:definitions name="MessageData"
   targetNamespace="http://bulletin.gt3.grid.itso.ibm.com/common/stubs"
   xmlns:tns="http://bulletin.gt3.grid.itso.ibm.com/common/stubs"
   xmlns:wsdl="http://schemas.xmlsoap.org/wsdl/">

<wsdl:types>
<schema targetNamespace="http://bulletin.gt3.grid.itso.ibm.com/common/stubs"
   attributeFormDefault="qualified"
   elementFormDefault="qualified"
   xmlns="http://www.w3.org/2001/XMLSchema">

   <complexType name="MessageDataType">
      <sequence>
         <element name="message" type="string"/>
      </sequence>
   </complexType>

</schema>
</wsdl:types>

</wsdl:definitions>
```

Figure 7-14 Message data type XML data specification

As cen be seen, the MessageDataType.xsd file defines a data structure called MessageData which contains a single String. This data structure will be used for storing the service data which, in this case, will contain only messages to be delivered to the subscriber clients. Including these definitions in our GWSDL file causes a Java Bean class to be automatically generated along with the stubs. This Java Bean class will then be available for the coding of the service data, as we will see shortly.

Figure 7-15 on page 140 shows another extract of the GWSD file, highlighting the service data declaration.

```
<gwsdl:portType name="BulletinPortType" extends="ogsi:GridService ogsi:NotificationSource">
  <operation name="...
...
  </operation>
  <sd:serviceData name="ApprovedMessages"
                  type="data:MessageDataType"
                  minOccurs="1"
                  maxOccurs="1"
                  mutability="mutable"
                  modifiable="false"
                  nillable="false">
  </sd:serviceData>
</gwsdl:portType>
```

Figure 7-15 Segment of the revised GWSDL showing the addition of service data declaration

As shown, the type for this service data is referred to as data:MessageDataType. The *data* tag is bound to the namespace defined for our data structure, and MessageDataType is its name. The several additional fields define the properties of this service data.

Once the GWSDL file has been updated, we can generate the stubs and move to the implementation of the service and its clients.

Implementation of the BulletinOprImpl class

As will be shown shortly, the BulletinOprImpl class additionally implements the GridServiceCallback interface (Figure 7-16 shows the definition of this interface). The interface defines callback methods that have to be implemented by grid services that need a more accurate control of their life cycle. These methods are invoked by the container runtime at the appropriate life cycle moment of the grid service.

```
package org.globus.ogsa;
public interface GridServiceCallback {
public void preCreate(GridServiceBase base) throws GridServiceException;
public void postCreate(GridContext context) throws GridServiceException;
public void activate(GridContext context) throws GridServiceException;
public void deactivate(GridContext context) throws GridServiceException;
public void preDestroy(GridContext context) throws GridServiceException;
}
```

Figure 7-16 Definition of the GridServiceCallback interface

The preCreate() method is invoked when the service instance is being created by the runtime and before it is fully created and initialized. The postCreate() is called after the service instance has been created and all of its configuration has been set up. The activate() method is called when a service becomes active according to the container's policy (a service is always activated before any of its methods are called). The deactivate() method is also a container's managed call which is normally fired in order to decrease the amount of resources taken by the grid server (memory, data-base connections, etc.). A deactivated service is, however, still discoverable by clients. The preDestroy() method is called just before a service is destroyed. After this call is made, the framework removes all knowledge about the service, so it is a good place to clean up service resources and store its state in persistent storage systems. Note that this call can be triggered by a client-initiated destroy call, as well as by a framework-initiated soft state time-out. It should also be noted that the parameter for all method calls, except preCreate(), is GridContext. The parameter for the preCreate() method is GridServiceBase. This is because the grid context for the service instance is not established until the instance is fully created and initialized.

Figure 7-17 on page 142 shows the first segment of the BulletinOprImpl class implementation. As can be seen from the figure, several new import statements are added. The first highlighted import statement imports the implementation of the MessageDataType class. This class is auto-generated from the GWSDL and provides the implementation of the class that encapsulates the bulletin message submitted by the writer. Similarly, definitions of ServiceData, GridContext, and GridServiceCallback are imported from the org.globus.ogsa package. The definition of the RemoteException is also imported. The use of these definitions will be evident as we provide further explanations.

```
package com.ibm.itso.grid.gt3.bulletin.server;

import com.ibm.itso.grid.gt3.bulletin.common.stubs.MessageDataType;
import org.globus.ogsa.ServiceData;
import org.globus.ogsa.GridContext;
import org.globus.ogsa.GridServiceBase;
import org.globus.ogsa.GridServiceException;
import org.globus.ogsa.OperationProvider;
import org.globus.ogsa.GridServiceCallback;
import javax.xml.namespace.QName;
import java.rmi.RemoteException;

public class BulletinOprImpl implements OperationProvider, GridServiceCallback {

   private MainSrvImpl myMain;
   private MessageDataType mdt;
   private String instanceName;
   private ServiceData approvedMessagesSDE;

   // Operation provider properties
   private static final QName[] operations = new QName[]{new QName("", "*")};
   private GridServiceBase base;
```

Figure 7-17 Implementation of the BulletinOprImpl class

As can be seen from Figure 7-17, the BulletinOprImpl additionally implements the GridServiceCallback interface. Several additional private variables are defined in this class. The myMain maintains a reference to the instance of the MainSrvImpl class. As we will illustrate shortly, this instance is created and maintained to obtain the system-defined Service Data Elements of the bulletin service. In particular, in our example, we need to access the name of the bulletin instance within the implementation of the service operations. The instance name is stored within a Service Data Element defined for all grid services and maintained by the container at runtime. The mdt object is an instance of a Java bean that has been created to store all the data that is sent along with the notifications. More details on this class will be given later. The instanceName variable maintains the name of the instance and the approvedMessageSDE maintains a reference to the Service Data Element that is defined for storing the bulletin messages submitted by the writer.

Figure 7-18 shows the continuation of the implementation of the BulletinOprImpl class. Additional code developed in this phase is highlighted.

```
public void initialize(GridServiceBase base) throws GridServiceException {
       this.base = base;
   }
       public QName[] getOperations() {
       return operations;
   }

   BulletinOprImpl(MainSrvImpl main) {
      myMain = main;
   }
public void preCreate(GridServiceBase arg0) throws GridServiceException {
       // Do nothing
   }

    public void postCreate(GridContext arg0) throws GridServiceException {
      instanceName = myMain.getInstanceName();
      System.out.println("Instance "+instanceName+" created.");

      approvedMessagesSDE = base.getServiceDataSet().create("ApprovedMessages");
      mdt = new MessageDataType();
      approvedMessagesSDE.setValue(mdt);
      mdt.setMessage("Initialized");
      base.getServiceDataSet().add(approvedMessagesSDE);
   }

   public void activate(GridContext arg0) throws GridServiceException {
      // Do nothing
   }
   public void deactivate(GridContext arg0) throws GridServiceException {
      // Do nothing
   }
public void preDestroy(GridContext arg0) throws GridServiceException {
       // Do nothing
   }
```

Figure 7-18 Implementation of the BulletinOprImpl class - continued

As can be seen from Figure 7-18, the BulletinOprImpl class implements a constructor method where an instance of the MainSrvImpl class is passed. Later in this section, we will present the implementation of the MainSrvImpl class and the creation of the BulletinOprImpl instance. It will suffice for now to note that an instance of the MainSrvImpl class provides access to metadata of the bulletin service instance, including the name of the service instance.

The BulletinOprImpl class also implements all the methods introduced in the GridServiceCallback interface. All the callback methods except postCreate() are empty. The postCreate() method performs two major tasks. First, it obtains and maintains the name of the service instance. Second, it creates and initializes the Service Data Element (SDE) for storing the messages that are submitted by the writer clients. Since this method is invoked right after the bulletin grid service instance is created and before any operation on it is invoked by the client, it is the most suitable place to perform the initialization of the SDE.

As can be seen in the implementation of the postCreate() method, the name of the instance is obtained by invoking the getInstanceName() method on the MainSrvImpl instance and this value is stored in a private String. Additionally, the Service Data Elements created in this method must conform to the previously generated GWSDL specification; as can be seen in the implementation, SDEs are created by the Service Data Set, which is in turn acquired from the GridServiceBase object, and must have the same names declared in the GWSDL file. In phase two, all bulletin messages submitted by the writer are automatically approved to be published to the subscribers.

Subsequent to the creation of the SDE structure, an instance of the MessageDataType is created and associated with the SDE by invoking the setValue() method on the SDE. At this point, the MessageDataType instance is a place holder for placing the messages that will be submitted by the writer. The SDE created is added into the service data set by invoking the add() method on it and by passing the newly created SDE. The runtime manages the SDE.

Figure 7-19 shows the final segment of the implementation of the BulletinOprImpl class. The submitApprovedMessages() method is the implementation of the **submitApprovedMessages** operation which was defined for the bulletin service in the GWSDL described earlier in this chapter. When the writer client invokes the operation, the server runtime will call this method in the BulletinOprImpl class.

```
    public void submitApprovedMessages(java.lang.String msgs) throws RemoteException {
        System.out.println("Instance "+instanceName+" received message: " +msgs);
        mdt.setMessage(instanceName+": "+msgs);
        approvedMessagesSDE.notifyChange();
    }
}
```

Figure 7-19 Implementation of the BulletinOprImpl class - continued

As shown in Figure 7-19, the submitApprovedMessages() method receives a string with the bulletin message. The implementation of the method performs two main tasks. First, it sets the message in the MessageDataType placeholder in

the SDE by invoking the setMessage() method. This places the writer's message in the approvedMessagesSDE. Second, it triggers the notification mechanism since the state of the approvedMessageSDE has been modified. The notification mechanism is triggered by invoking the notifyChange() method on the SDE being modified. When notifyChange() is invoked, the runtime notifies all the subscribers who have registered an interest in changes to the SDE that the SDE has been modified.

Implementation of the MainSrvImpl class

In the previous section, the methods of the BulletinOprImpl class maintained and used an instance of the MainSrvimpl class. The instance was mainly used to obtain the name of the bulletin grid service instance. In this section, we will illustrate the implementation of the MainSrvImpl class.

During the first phase shown in 7.4.2, "Administration client implementation" on page 132, the implementation of the administrator client was discussed. The administrator client created instances of the bulletin service by providing the name of the service instance. The name of the instance becomes part of the metadata of the service instance after creation. The GridServiceImpl class provided at runtime maintains all the metadata in a set of standard Service Data Elements. Hence, in order to obtain the instance name, we implement a new class called MainSrvImpl which inherits from the GridServiceImpl class and provides a public method that encapsulates the code to obtain the name of the instance. Figure 7-20 on page 146 shows the implementation of the MainSrvImpl class.

```
package com.ibm.itso.grid.gt3.bulletin.server;

import org.globus.ogsa.GridServiceException;
import org.globus.ogsa.impl.ogsi.GridServiceImpl;
import org.gridforum.ogsi.LocatorType;
import javax.xml.namespace.QName;
import org.globus.ogsa.ServiceData;
import org.globus.ogsa.ServiceDataSet;

public class MainSrvImpl extends GridServiceImpl {

        public MainSrvImpl() throws GridServiceException {
                super("Bulletin Service Implementation");
                this.addOperationProvider(new BulletinOprImpl(this));
        }

public String getInstanceName() {
                String Space = "http://www.gridforum.org/namespaces/2003/03/OGSI";
                try {
                        ServiceDataSet dataset = this.getServiceDataSet();
                        QName handle = new QName(Space,"gridServiceHandle");
                        QName factoryLocator = new QName(Space,"factoryLocator");
                        ServiceData dataOfHandle = dataset.get(handle);
                        ServiceData dataOfFactor = dataset.get(factoryLocator);

                        String handleS = dataOfHandle.getValue().toString();

                        LocatorType locatortype = (LocatorType)dataOfFactor.getValue();
                        String factorS= (locatortype.getHandle()[0]).getValue().toString();

                        int lastPosition = handleS.lastIndexOf("/");
                        if (factorS.equalsIgnoreCase(handleS.substring(0,lastPosition  ))) {
                                String InstanceName = handleS.substring(lastPosition +1,
handleS.length());
                                System.out.println("Get instance Name:"+InstanceName);
                                return InstanceName;
                        }
                } catch (Exception e) {
                        e.printStackTrace();
                        return null;
                }
                return null;
        }
}
```

Figure 7-20 Implementation of the MainSrvImpl class

As shown in Figure 7-20 on page 146, the MainSrvImpl class provides two public methods. The first method is the constructor, which is invoked when an instance of MainSrvImpl is created. The second is the getInstanceName() method, which returns a string containing the name of the instance. Whenever the client of the bulletin service (in our case the administrator client) creates the instance of the bulletin service, the container creates an instance of the MainSrvI.mpl class.

The constructor of the MainSrvImpl class first invokes the constructor of the GridServiceImpl class by calling super(). Subsequently, it creates an instance of the BulletinOprImpl and adds it to its the metadata by invoking the addOperationProvider() method. It should be noted that the addOperationProvider() method is inherited by the MainSrvImpl class from the GridServiceImpl class. In addition, the reference to the MainSrvImpl instance is passed to the BulletinOprImpl instance during its creation.

The getInstanceName() method derives the name of the instance from the metadata stored in the service data set. Specifically, it derives the instance name from the string representing the two key Service Data Elements, namely, the gridServicehandle and the factoryLocator. The service data set is first obtained by invoking the getServiceDataSet() method. Subsequently, individual Service Data Elements for the gridServicehandle and the factoryLocator are obtained from the service data set by invoking the get() method and passing the QName for them. Finally, their String representation is obtained and the instance name is derived by parsing it out of the gridServiceHandle string.

In summary, in this section we have illustrated the following key mechanisms:

1. The implementation of the BulletinOprImpl class for supporting the **`submitApprovedMessages`** operation of the bulletin grid service.
2. The creation and use of the Service Data Element (SDE) to store the message submitted by the writer.
3. The mechanism to fire notifications when a new bulletin message is stored in the SDE.
4. The implementation of the MainSrcImpl class to obtain the name of the instance.

7.5.2 Writer client implementation

In this section, we will implement the writer client. As mentioned previously, the writer client provides a user interface for the writer to submit his/her bulletin messages. The writer client is a client of the bulletin service whose specification was discussed in the previous section. The writer can submit bulletin messages to any of the instances of the bulletin service that have been created by the administrator client and, as was shown in the previous phase, one instance of

the bulletin service is created for each of the topics chosen. The topic is stored as the name of the service instance. Examples of topics are Business News, Weather, and Sports.

Figure 7-21 shows the first segment of the implementation of the writer client. The implementation of the writer client is provided by the class BulletinWriter. The various import statements are shown in Figure 7-21. As with the implementation in other clients previously shown, the BulletinServiceGridLocator class aids in resolving the GSH into a GSR for a given bulletin grid service instance. A variable of BulletinPortType maintains the client reference to the bulletin grid service instance. The use of other imports will be obvious as we show more of the implementation.

```
package com.ibm.itso.grid.gt3.bulletin.client;

import com.ibm.itso.grid.gt3.bulletin.common.stubs.service.BulletinServiceGridLocator;
import com.ibm.itso.grid.gt3.bulletin.common.stubs.BulletinPortType;

import java.io.BufferedReader;
import java.io.InputStreamReader;
import java.net.URL;
import java.util.List;
import java.util.ArrayList;
```

Figure 7-21 Implementation of the BulletinWriter class

```
public class BulletinWriter {
   public static final void main(String[] args) {
      // The base GSH and the instance names will be informed
      // as command-line arguments
      String baseGSH = args[0];
      // Build service instance references
      int instanceCount = args.length-1;
      BulletinPortType[] bulletins = new BulletinPortType[instanceCount];
      List instanceNames = new ArrayList(instanceCount);
      try {
         for (int i=0; i<instanceCount; i++) {
            String GSHstr = baseGSH+"/"+args[1+i];
            URL GSH = new URL(GSHstr);
            BulletinServiceGridLocator bulletinGL = new BulletinServiceGridLocator();
            bulletins[i] = bulletinGL.getBulletinServicePort(GSH);
            instanceNames.add(args[i+1]);
         };
         boolean go = true;
         while (go) {
            BufferedReader br
                    = new BufferedReader(new InputStreamReader(System.in));
            System.out.println("Enter the type of message you want to submit");
            String instanceName = br.readLine();
            int instanceIndex = instanceNames.indexOf(instanceName);
            if (instanceIndex == -1) {
               System.out.println("There isn't any service instance named "+instanceName);
            } else {
               System.out.println("Enter the message");
               String message = br.readLine();

               bulletins[instanceIndex].submitApprovedMessages(message);
            }
            System.out.println("Do you want to submit another message (y/n) ?");
            char key = Character.toLowerCase((char) br.read());
            if (key != 'y') {
               go = false;
            }
         }
      } catch (Exception e) {
         e.printStackTrace();
      }
   }
}
```

Figure 7-22 Implementation of the BulletinWriter class - continued

Figure 7-21 on page 148 is the continuation of the WriterBulletin class implementation. The WriterBulletin receives the base GSH of the bulletin grid service instances as the first argument. The name of all the topics that the News Service is providing service for and hence the writer can submit messages to are passed as subsequent arguments. As a first step, the WriterBulletin generates the GSHs of all the relevant bulletin grid service instances based on the argument data. Subsequently, it resolves them into GSRs and maintains these references for future use. The bulletins array is used to store the GSRs. The names of the instances or the news topics are stored in the instanceNames array list. The key lines of code which generate each of the GSH and resolve them into GSRs are highlighted. An instance of BulletinServiceGridLocator is created and this aids in the resolution of the GSH. The getBulletinServicePort() method is invoked on the locator and the GSH is passed for resolution. The method returns the corresponding GSR of a bulletin grid service instance.

After obtaining the references of the bulletin grid service instances, the WriterBulletin allows the writer to interactively submit messages on the various topics. The implementation allows the writer to first enter the topic of the message followed by the news bulletin message itself. It checks the topic name entered to ensure that the topic corresponds to one of the bulletin service instances present in the server. If the topic is indeed among the ones present in the server, the user is allowed to enter the message. The message is submitted by invocation of the **submitApprovedMessages** operation on the corresponding bulletin grid service instance. The key line of code performing the service invocation is highlighted in bold. The writer is allowed to continuously submit messages until he/she decides to quit the writer client application.

7.5.3 Enhancing the subscriber client implementation

In this section, we will expand the subscriber client implementation from phase one to allow the subscriber to be notified of the news bulletins that are submitted by the writer. In phase one, the subscriber client, implemented by class BulletinSubscriber, initialized and obtained references to the various bulletin service instances. In the previous sub-sections of this phase, the writer submitted the messages to the appropriate bulletin service instance. Also, the implementation of **submitApprovedMessages** at the server triggered the notification of changes to the SDE when a new bulletin message was submitted by the writer and stored in the SDE. In order for the subscriber to be notified of the messages, it must register for the notification. Thus, when the server triggers the notification, the runtime delivers the added messages to the entity that registered for it, namely to the subscriber.

Figure 7-23 on page 151 shows the first segment of the BulletinSubscriber client which implements the subscriber client. The newly added code in this phase is highlighted. The use of the MessageDataType for the various messages is the

same as explained previously. The imports from the org.globus.ogsa package provide the runtime functionality needed to support the registration and delivery of the notification, as will become evident shortly. The use of other imports is similar to that of other implementations explained earlier.

```
package com.ibm.itso.grid.gt3.bulletin.client;

import com.ibm.itso.grid.gt3.bulletin.common.stubs.service.BulletinServiceGridLocator;
import com.ibm.itso.grid.gt3.bulletin.common.stubs.BulletinPortType;
import com.ibm.itso.grid.gt3.bulletin.common.stubs.MessageDataType;

import org.globus.ogsa.utils.AnyHelper;
import org.globus.ogsa.NotificationSinkCallback;
import org.globus.ogsa.impl.core.service.ServicePropertiesImpl;
import org.globus.ogsa.client.managers.NotificationSinkManager;

import org.gridforum.ogsi.HandleType;
import org.gridforum.ogsi.ExtensibilityType;
import org.gridforum.ogsi.ServiceDataValuesType;

import java.net.URL;
import java.rmi.RemoteException;
import java.util.List;
import java.util.ArrayList;
public class BulletinSubscriber extends ServicePropertiesImpl implements
NotificationSinkCallback {

   List instanceNames;
   BulletinPortType[] bulletins;

   public static final void main(String[] args) {
      BulletinSubscriber bs = new BulletinSubscriber();
      bs.run(args);
   }
```

Figure 7-23 Implementation of the BulletinSubscriber class

As can be seen in Figure 7-23, the BulletinSubscriber class in this phase additionally extends the ServicePropertiesImpl class and implements the NotificationSinkCallback interface. In order to be notified and receive the notification messages, the BulletinSubscribe class must implement the NotificationSinkCallback interface and extend the ServicePropertiesImpl class. Briefly speaking, this class provides the basic functionality that a client needs to be able to receive notifications (such as a NotificationManager object). The instanceNames variable maintains the list of names of bulletin service instances or the news topics that the subscriber is interested in. As was explained in the

earlier phase, the names of the various news topics are provided as argument values when the BulletinSubscriber is started.

Figure 7-24 on page 153 shows the continuation of the implementation of the BulletinSubscriber class. In addition to creating the GSH for the bulletin grid service instances and resolving their GSRs as was done in the previous phase, the implementation of the run() method also maintains a list of names of the bulletin service instances in the instanceNames variable.

In order to receive the messages that are stored in the ApprovedMessages SDE of the bulletin grid service instance, the subscriber client has to register for the notifications. This registration is accomplished by adding a listener to the SDE. As can be seen in Figure 7-24 on page 153, this is accomplished with the aid of the NotificationSinkManager instance. This is obtained by calling the getManager() static method of the NotificationSinkManager class. It returns the singleton NotificationSinkManager instance. The listening process is started by invoking the startListening() method on the NotificationSinkManager instance and passing the identifier of the waiting thread.

Finally, the listener is added to the SDE by invoking the addListner() method on the NotificationSinkManager instance. As shown in the figure, the name of the SDE, the handle to the grid service instance on which the listening is to be performed and the pointer to the NotificationSinkCallback have be to be passed along with the method invocation. In our case, the SDE is ApprovedMessages and the grid service instance is one of the bulletin grid service instances. Since the BulletinSubscriber class implements the NotificationSinkCallback interface and provides the implementation of the deliverNotification() callback method, as shown later, its reference can be passed to the addListner() method. It should be noted that addListner() is invoked inside the for loop and is invoked once for each of the bulletin grid service instances. Further, since the same reference of the NotificationSinkCallback instance ‘this’ is passed in with multiple invocations of the addListner, the deliverNotification() method of this instance will receive notifications from the SDEs of multiple bulletin service instances. After adding the listeners, the implementation of the run() method waits for user input before exiting.

```
private void run(String[] args) {

      // The base GSH and the instance names will be informed
      // as command-line arguments
      String baseGSH = args[0];

      // Build service instance references
      int instanceCount = args.length-1;
      BulletinPortType[] bulletins = new BulletinPortType[instanceCount];
      instanceNames = new ArrayList(instanceCount);

      try {
         NotificationSinkManager notifManager = NotificationSinkManager.getManager();
         String sinks[] = new String[instanceCount];

         for (int i=0; i<instanceCount; i++) {
            String instanceName = args[i+1];
            String GSHstr = baseGSH+"/"+instanceName;
            URL GSH = new URL(GSHstr);
            BulletinServiceGridLocator bulletinGL = new BulletinServiceGridLocator();
            bulletins[i] = bulletinGL.getBulletinServicePort(GSH);

            notifManager.startListening(NotificationSinkManager.MAIN_THREAD);
            sinks[i] = notifManager.addListener("ApprovedMessages", null, new
HandleType(GSHstr), this);

            instanceNames.add(instanceName);
            System.out.println("Subscribed to the \""+instanceName+"\" instance.");
         };

         System.out.println("Type any key to exit.");
         System.in.read();

         // Stop listening
         for (int i=0; i<instanceCount; i++) {
            notifManager.removeListener(sinks[i]);
         }

         notifManager.stopListening();
         System.out.println("Not listening anymore!");
      } catch (Exception e) {
         e.printStackTrace();
      }
   }
```

Figure 7-24 Implementation of the BulletinSubscriber class - continued

As shown in Figure 7-24 on page 153, while exiting, the run() method removes the listeners and stops listening for the notifications. When a listener is added, a handle to the listener is returned by the runtime. It is stored in the sinks array. The listener is removed by invoking the removeListner() method on the NotificationSinkManager and by passing the handle that was obtained while adding the listener. The listening is stopped by invoking the stopListening() method on the NotificationSinkManager.

Whenever a bulletin message is submitted by the writer, the server triggers a change notification. The listener detects the notification event and the notification is delivered to the class implementing the NotificationSinkCallback interface. As shown above, in our example the BulletinSubscriber class itself implements the NotificationSinkCallback interface, which contains the deliverNotification() method. Figure 7-25 shows the implementation of this method.

```
public void deliverNotification(ExtensibilityType any) throws RemoteException {
       ServiceDataValuesType serviceData = AnyHelper.getAsServiceDataValues(any);
       MessageDataType mdt = (MessageDataType) AnyHelper.getAsSingleObject(serviceData,
MessageDataType.class);

       System.out.println("New message received.\n"+mdt.getMessage());
    }
}
```

Figure 7-25 Implementation of the BulletinSubscriber class - continued

As shown in Figure 7-25, when deliverNotification() is called, the service data is passed in the any variable. The method extracts the notification data from the service data and presents it to the Subscriber. This is a push notification whereby the service data is pushed to the subscriber rather than the subscriber merely being informed of the change in service data and having to fetch or pull the data from the bulletin service. The static methods of the AnyHelper class provide the utility functions needed for extracting the service data and the bulletin message from the service data. The getAsServiceDatavalues() method extracts the service data from the incoming any variable. The getAsSingleObject() obtains the message of the MessageDataType class from the service data. The method is passed the class object of the message along with service data.

7.6 Phase III: incorporating workflow and approval by editor

In the third phase, we will expand the News Service implementation of the second phase, incorporating a workflow within the News Service. In phase two,

any message that was submitted by the writer was immediately made available to the subscriber via push notification. In this phase, we introduce an additional actor, the editor, who is responsible for approving the news bulletin messages before they are pushed to the subscribers. Hence, any messages that are submitted by the writer will be held pending until the editor approves them. Only after a message has been approved by the editor will it be pushed to the subscriber.

Here we will focus on supporting the use case(s) related to the editor. We will implement the **Register for Pending Bulletin**, **Retrieve Pending Bulletin**, and **Approve Bulletin** use cases. In addition, we will support the **Notify Pending Bulletin Existence** use case which the News Service itself triggers. Thus, the writer will register for pending messages that are submitted by the writer. The bulletin service will notify the editor of the existence of pending bulletin news messages. The editor will pull or retrieve the pending bulletin messages from the service instance(s) for revision, and will approve them or not. The approved messages are pushed to the subscriber as in the second phase.

As we had specified in the architectural decisions in the previous chapter, the following actions will take place:

1. Implement the editor user interface as a single editor client module.
2. Add a new service data to the bulletin service port type. The GWSDL file has to be modified in the same manner as in the last phase, with the exception that this additional service data will make use of the same data structure we have already introduced for the ApprovedMessages Service Data Element.
3. Implement the bulletin grid service operations to be supported for the different actors in separate operation provider classes. Thus, we will implement the **submitMessageForApproval** operation in one operation provider, and implement **getPendingMessages** and **submitApprovedMessages** operations in another operation provider. The first class manages pending messages whereas the second class manages approved messages.
4. Add an additional SDE to the bulletin service implementation called PendingMessagesSDE.
5. The writer will be modified to invoke **submitMessageForApproval** operation instead of the **submitApprovedMessages** as in the previous phase. The ApprovedMessages SDE and the **SubmitApprovedMessages** already implemented in the previous phase will be used by the editor client. All approved messages submitted by the editor will be stored in the ApprovedMessagesSDE.

Support of the aforementioned functionality requires the following grid services features:

1. The use of the multiple Service Data Element (SDE) in the same grid service.
2. The implementation of the multiple Operation Provider classes for the same service.
3. The implementation of the Pull Notification in supporting the notification of service state changes.

7.6.1 Enhancing the server side functionality

In this section, we will illustrate the code on the server side, with particular emphasis on the changes and additions that were made to support the functionality for the third phase. In particular, we will introduce two operation provider classes, namely BulletinPenMesgOprImpl and BulletinAppMesgOprImpl, replacing the BulletinOprImpl class that was implemented in phase two. The BulletinPenMesgOprImpl class will implement the **`getPendingMessages`** and the **`submitMessageForApproval`** operations. The BulletinAppMesgOprImpl will implement the **`submitApprovedMessages`** operation. In addition, we will show the modifications to the implementation of the MainSrvImpl class. The constructor of the MainSrvImpl class in this phase will create instances of the two operation provider classes instead of the one operation provider class as in the previous phase. Further, we will create an additional SDE, PendingMessagesSDE, which will be used to store messages that are submitted by the writer. This is in addition to the ApprovedMessagesSDE which was created in the previous phase. Unlike in the previous phase where it was used for storing the messages submitted by the writer, in this phase it will be used for storing the messages that are approved by the editor.

Implementation of the BulletinPenMesgOprImpl class

In this section, we will illustrate the implementation of the BulletinPenMesgOprImpl class. As mentioned before, this class is one of the operation provider classes of the bulletin grid service. It implements the **`getPendingMessages`** and the **`submitMessageForApproval`** operations. Figure 7-26 on page 157 shows the implementation of the class.

The specification and use of the various imports statements are the same as explained in the implementation of the BulletinOprImpl class in phase two. The need of the operation provider class to implement the initialize() and getOperations() methods was also explained. This class will create and use the PendingMessageSDE Service Data Element. Further, the class will store all the

messages that are submitted by the writer and not retrieved by the editor in the pendingMessageChunk private variable.

```
package com.ibm.itso.grid.gt3.bulletin.server;

import com.ibm.itso.grid.gt3.bulletin.common.stubs.MessageDataType;

import org.globus.ogsa.GridContext;
import org.globus.ogsa.ServiceData;
import org.globus.ogsa.GridServiceBase;
import org.globus.ogsa.GridServiceException;
import org.globus.ogsa.OperationProvider;
import org.globus.ogsa.GridServiceCallback;

import javax.xml.namespace.QName;
import java.rmi.RemoteException;

public class BulletinPenMesgOprImpl implements OperationProvider, GridServiceCallback {

   private MainSrvImpl myMain;

   private MessageDataType mdt;

   private String instanceName;
   private String pendingMessageChunck = "";

   private ServiceData pendingMessageSDE;

   // Operation provider properties
   private static final QName[] operations = new QName[]{new
QName("http://bulletin.gt3.grid.itso.ibm.com/common/stubs", "submitMessageForApproval"),
                          new QName("http://bulletin.gt3.grid.itso.ibm.com/common/stubs",
"getPendingMessages")};
   private GridServiceBase base;

      public void initialize(GridServiceBase base) throws GridServiceException {
      this.base = base;
   }
   public QName[] getOperations() {
      return operations;
   }

   BulletinPenMesgOprImpl(MainSrvImpl main) {
      myMain = main;
   }
```

Figure 7-26 Implementation of the BulletinPenMesgOprImpl class

As was mentioned in phase two, this class implements the GridServiceCallback interface in addition to the OperationProvider class. Figure 7-27 is the continued implementation of the BulletinPenMesgOprImpl class and shows the implementation of the methods introduced in the GridServiceCallback interface. Just as was explained in phase two, this class implements the postCreate() method to initialize the use of the service data. All other callback methods have empty implementations.

The postCreate() method is invoked by the container at runtime soon after the instance is created and before any grid service operation is dispatched. The method creates a Service Data Element called PendingMessages by calling the create() method on the service data set of the bulletin service. The service data set is obtained by invoking the getServiceDataSet() method on the GridServiceBase instance. The message is stored in an instance of type MessageDataType as explained in phase two and added to the service data by invoking the setValue() method.

```
public void preCreate(GridServiceBase arg0) throws GridServiceException {
       // Do nothing
   }

public void postCreate(GridContext arg0) throws GridServiceException {
       instanceName = myMain.getInstanceName();
       System.out.println("Instance "+instanceName+" created.");

       pendingMessageSDE = base.getServiceDataSet().create("PendingMessages");

       mdt = new MessageDataType();
       pendingMessageSDE.setValue(mdt);
       mdt.setMessage(instanceName);

       base.getServiceDataSet().add(pendingMessageSDE);
   }
       public void activate(GridContext arg0) throws GridServiceException {
       // Do nothing
   }
       public void deactivate(GridContext arg0) throws GridServiceException {
       // Do nothing
   }
   public void preDestroy(GridContext arg0) throws GridServiceException {
       // Do nothing
   }
```

Figure 7-27 Implementation of the BulletinPenMesgOprImpl class - continued

Figure 7-28 on page 159 is the continuation of the implementation of the BulletinPenMesgOprImpl class. The implementation of the methods that correspond to the bulletin service operations is shown. The

submitMessageForApproval() method is invoked by the writer when submitting a new bulletin message. The method appends the submitted message to a private variable and triggers a notification message to the clients that have registered an interest in the pending message, namely the editor. The runtime delivers the notification to the registered parties.

```
public void submitMessageForApproval(java.lang.String msg) throws RemoteException {
   System.out.println("Instance "+instanceName+" received message: " +msg);
   pendingMessageChunck += msg + "\n";

   pendingMessageSDE.notifyChange();
}

   public java.lang.String getPendingMessages() throws RemoteException {
   String result = new String(pendingMessageChunck);
   pendingMessageChunck = "";
   return result;
}
   }
```

Figure 7-28 Implementation of the BulletinPenMesgOprImpl class - continued

After receiving the notification of new pending messages, the editor invokes the getPendingMessages() method in order to retrieve the bulletin messages that have been submitted by the writer. The implementation of the method returns the pending messages since the last time the editor invoked this method. The method also resets the pendingMessageChunck to an empty string. This is to ensure that the next time the editor obtains the pending messages, no previously extracted messages are received.

Implementation of the BulletinAppMesgOprImpl class

In this section, we will illustrate the implementation of the BulletinAppMesgOprImpl class. As mentioned before, this class is one of the operation provider classes of the bulletin grid service and is responsible for implementing the **`submitMessageForApproval`** operations. The set of figures below shows the implementation of the class. As can be seen from the figures, the implementation is identical to that of the BulletinOprImpl class shown in phase two. The only difference is in the name of the class. In order to make the name meaningful, we changed the name from BulletinOprImpl to BulletinAppMesgOprImpl.

```
package com.ibm.itso.grid.gt3.bulletin.server;

import com.ibm.itso.grid.gt3.bulletin.common.stubs.MessageDataType;

import org.globus.ogsa.GridContext;
import org.globus.ogsa.GridServiceCallback;
import org.globus.ogsa.ServiceData;
import org.globus.ogsa.GridServiceBase;
import org.globus.ogsa.GridServiceException;
import org.globus.ogsa.OperationProvider;

import java.rmi.RemoteException;

import javax.xml.namespace.QName;

public class BulletinAppMesgOprImpl implements OperationProvider, GridServiceCallback {

   private MainSrvImpl myMain;

   private MessageDataType mdt;

   private String instanceName;

   private ServiceData approvedMessagesSDE;

   // Operation provider properties
   private static final QName[] operations = new QName[]{new
QName("http://bulletin.gt3.grid.itso.ibm.com/common/stubs", "submitApprovedMessages")};
   private GridServiceBase base;

      public void initialize(GridServiceBase base) throws GridServiceException {
      this.base = base;
   }

      public QName[] getOperations() {
      return operations;
   }

   BulletinAppMesgOprImpl(MainSrvImpl main) {
      myMain = main;
   }
```

Figure 7-29 Implementation of the BulletinAppMesgOprImpl Class

```
    public void preCreate(GridServiceBase arg0) throws GridServiceException {
       // Do nothing
    }
    public void postCreate(GridContext arg0) throws GridServiceException {
       instanceName = myMain.getInstanceName();
       System.out.println("Instance "+instanceName+" created.");

       approvedMessagesSDE = base.getServiceDataSet().create("ApprovedMessages");

       mdt = new MessageDataType();
       approvedMessagesSDE.setValue(mdt);
       mdt.setMessage("Initialized");

       base.getServiceDataSet().add(approvedMessagesSDE);
    }

       public void activate(GridContext arg0) throws GridServiceException {
       // Do nothing
    }

       public void deactivate(GridContext arg0) throws GridServiceException {
       // Do nothing
    }

       public void preDestroy(GridContext arg0) throws GridServiceException {
       // Do nothing
    }
public void submitApprovedMessages(java.lang.String msgs) throws RemoteException {
       mdt.setMessage(instanceName+": "+msgs);
       approvedMessagesSDE.notifyChange();
    }
}
```

Figure 7-30 Implementation of the BulletinAppMesgOprImpl class - continued

Implementation of the MainSrvImpl class

Figure 7-31 on page 162 shows the implementation of the MainSrvImpl Class. The additions to the implementation from phase two are highlighted. Since in phase three, we implement two operation provider classes for the bulletin grid service, the constructor of the MainSrvImpl has to create instances of those classes and add them to the grid service implementation.

```
package com.ibm.itso.grid.gt3.bulletin.server;

import org.globus.ogsa.GridServiceException;
import org.globus.ogsa.impl.ogsi.GridServiceImpl;
import org.gridforum.ogsi.LocatorType;
import javax.xml.namespace.QName;
import org.globus.ogsa.ServiceData;
import org.globus.ogsa.ServiceDataSet;

public class MainSrvImpl extends GridServiceImpl {

        public MainSrvImpl() throws GridServiceException {
                super("Bulletin Service Implementation");
this.addOperationProvider(new BulletinAppMesgOprImpl(this));
this.addOperationProvider(new BulletinPenMesgOprImpl(this));
        }
public String getInstanceName() {

                String Space = "http://www.gridforum.org/namespaces/2003/03/OGSI";
                try {
                        ServiceDataSet dataset = this.getServiceDataSet();
                        QName handle = new QName(Space,"gridServiceHandle");
                        QName factoryLocator = new QName(Space,"factoryLocator");
                        ServiceData dataOfHandle = dataset.get(handle);
                        ServiceData dataOfFactor = dataset.get(factoryLocator);

                        String handleS = dataOfHandle.getValue().toString();

                        LocatorType locatortype = (LocatorType)dataOfFactor.getValue();
                        String factorS= (locatortype.getHandle()[0]).getValue().toString();

                        int lastPosition = handleS.lastIndexOf("/");
                        if (factorS.equalsIgnoreCase(handleS.substring(0,lastPosition  ))) {
                                String InstanceName = handleS.substring(lastPosition +1,
handleS.length());
                                System.out.println("Get instance Name:"+InstanceName);
                                return InstanceName;
                        }
                } catch (Exception e) {
                        e.printStackTrace();
                        return null;
                }
                return null;
        }
}
```

Figure 7-31 Implementation of the MainSrvImpl class

7.6.2 Modifying the writer client

In this section, we present the implementation of the writer client and discuss the changes made to the writer client from phase two in order to incorporate the editor approval and the workflow in the submission, approval, and subscription notification of the bulletin messages.

Figure 7-32 shows the implementation of the writer client. The implementation of the writer client is identical to that of phase two except for one major difference.

```
package com.ibm.itso.grid.gt3.bulletin.client;

import com.ibm.itso.grid.gt3.bulletin.common.stubs.service.BulletinServiceGridLocator;
import com.ibm.itso.grid.gt3.bulletin.common.stubs.BulletinPortType;

import java.io.BufferedReader;
import java.io.InputStreamReader;
import java.net.URL;
import java.util.List;
import java.util.ArrayList;

public class BulletinWriter {

   public static final void main(String[] args) {
      // The base GSH and the instance names will be informed
      // as command-line arguments
      String baseGSH = args[0];

      // Build service instance references
      int instanceCount = args.length-1;
      BulletinPortType[] bulletins = new BulletinPortType[instanceCount];
      List instanceNames = new ArrayList(instanceCount);

      try {
         for (int i=0; i<instanceCount; i++) {
            String instanceName = args[i+1];
            String GSHstr = baseGSH+"/"+instanceName;
            URL GSH = new URL(GSHstr);
            BulletinServiceGridLocator bulletinGL = new BulletinServiceGridLocator();
            bulletins[i] = bulletinGL.getBulletinServicePort(GSH);
            instanceNames.add(instanceName);
         };
```

Figure 7-32 Implemenation of the BulletinWriter class

The difference in the implementation of the writer client from phase two to phase three is highlighted. In the second phase, the writer submitted the bulletin messages by invoking the submitApprovedMessages() and the messages were directly pushed to the subscribing client. However, in this phase, the workflow has been introduced and an additional actor, namely the editor, is responsible for approving the submitted messages. Hence, the writer submits the message by invoking the submitMessageForApproval operation.

```
          boolean go = true;
          while (go) {
             BufferedReader br
                     = new BufferedReader(new InputStreamReader(System.in));

             System.out.println("Enter the type of message you want to submit");

             String instanceName = br.readLine();
             int instanceIndex = instanceNames.indexOf(instanceName);

             if (instanceIndex == -1) {
                System.out.println("There isn't any service instance named "+instanceName);
             } else {
                System.out.println("Enter the message");
                String message = br.readLine();

                bulletins[instanceIndex].submitMessageForApproval(message);
             }

             System.out.println("Do you want to submit another message (y/n) ?");
             char key = Character.toLowerCase((char) br.read());
             if (key != 'y') {
                go = false;
             }
          }
       } catch (Exception e) {
          e.printStackTrace();
       }
    }
}
```

Figure 7-33 Implementation of the BulletinWriter class - continued

7.6.3 Implementing the editor client

In this section, we discuss the implementation of the editor client. As mentioned previously, the editor client provides a user interface for the editor to retrieve the pending bulletin messages that are submitted by the writer, review, and approve

them. The editor can retrieve, review, and approve the bulletin messages on any of the topics the News Service is currently interested in providing the service.

```
package com.ibm.itso.grid.gt3.bulletin.client;

import com.ibm.itso.grid.gt3.bulletin.common.stubs.service.BulletinServiceGridLocator;
import com.ibm.itso.grid.gt3.bulletin.common.stubs.BulletinPortType;
import com.ibm.itso.grid.gt3.bulletin.common.stubs.MessageDataType;

import org.globus.ogsa.utils.AnyHelper;
import org.globus.ogsa.NotificationSinkCallback;
import org.globus.ogsa.impl.core.service.ServicePropertiesImpl;
import org.globus.ogsa.client.managers.NotificationSinkManager;

import org.gridforum.ogsi.HandleType;
import org.gridforum.ogsi.ExtensibilityType;
import org.gridforum.ogsi.ServiceDataValuesType;

import java.io.BufferedReader;
import java.io.InputStreamReader;
import java.net.URL;
import java.rmi.RemoteException;
import java.util.List;
import java.util.Iterator;
import java.util.ArrayList;
import java.util.StringTokenizer;
```

Figure 7-34 Implementation of the BulletinEditor class

Figure 7-34 shows the first segment of the implementation of the editor client. The implementation of the editor client is provided by the class BulletinEditor. Similar to the implementation of other clients shown previously, the BulletinServiceGridLocator class aids in resolving the GSH into a GSR for a given bulletin service instance. A variable of BulletinPortType maintains the client reference to the bulletin grid service instance. An instance of the MessageDataType stores the bulletin messages that are submitted by the writer and reviewed by the editor. The use of the other imports is identical to that in the BulletinSubscriber client. For an explanation of the imports, refer to the description of the BulletinSubscriber implementation in phase two.

Figure 7-34 shows the continuation of the BulletinEditor class. Similar to the BulletinSubscriber class, the BulletinEditor class also extends the ServicePropertiesImpl class and implements the NotificationSinkCallback interface. As shown below, the entire logic of the BulletinEditor is enclosed in the run() method which we will discuss shortly.

```
public class BulletinEditor extends ServicePropertiesImpl implements
NotificationSinkCallback {

   List instanceNames;
   BulletinPortType[] bulletins;

   public static final void main(String[] args) {
      BulletinEditor be = new BulletinEditor();
      be.run(args);
   }
```

Figure 7-35 Implementation of the BulletinEditor class - continued

Figure 7-36 on page 167 is the continuation of the BulletinEditor class and shows the implementation of the run() method. When the editor client is invoked, the address of the bulletin service and the name(s) of the service instances that the new service is providing service for are passed as arguments. The implementation constructs the GSRs of the instance name(s) passed, in URL form, by concatenating each of the instance names with the address of the bulletin service.

Once the GSH of the bulletin service instances has been created, it resolves the references into GSRs. An instance of the BulletinServiceGridLocator instance class aids in resolving GSHs into GSRs. As shown in the code below, the editor client creates an instance of the BulletinServiceGridLocator class and invokes the getBulletinServicePort() method on it by passing the GSH parameter. Obtaining a reference to the service instances is critical to the support of other editor client functionality. The implementation of the run() method also maintains a list of names of the bulletin service instances in the instanceNames variable.

In order to receive the messages that are stored in the PendingMessages SDE of the bulletin service instance, the editor client has to register for the notifications. This registration is accomplished by adding a listener to the SDE. As can be seen in Figure 7-36 on page 167, this is accomplished with the aid of the NotificationSinkManager instance. This is obtained by calling the getManager() static method of the NotificationSinkManager class. It returns the singleton NotificationSinkManager instance. The listening process is started by invoking the startListenting() method on the NotificationSinkManager instance and passing the identifier of the waiting thread.

```
private void run(String[] args) {

      // The base GSH and the instance names will be informed
      // as command-line arguments
      String baseGSH = args[0];

      // Build service instance references
      int instanceCount = args.length-1;
      BulletinPortType[] bulletins = new BulletinPortType[instanceCount];
      instanceNames = new ArrayList(instanceCount);

      try {
         NotificationSinkManager notifManager = NotificationSinkManager.getManager();
         String sinks[] = new String[instanceCount];

         for (int i=0; i<instanceCount; i++) {
            String GSHstr = baseGSH+"/"+args[1+i];
            URL GSH = new URL(GSHstr);
            BulletinServiceGridLocator bulletinGL = new BulletinServiceGridLocator();
            bulletins[i] = bulletinGL.getBulletinServicePort(GSH);

            notifManager.startListening(NotificationSinkManager.MAIN_THREAD);
            sinks[i] = notifManager.addListener("PendingMessages", null, new
HandleType(GSHstr), this);
            instanceNames.add(args[i+1]);
         };
```

Figure 7-36 Implementation of the BulletinEditor class - continued

Finally, the listener is added to the SDE by invoking the addListner() method on the NotificationSinkManager instance. As shown in the figure, the name of the SDE, the handle to the service instance on which the listening is to be performed and the pointer to the NotificationSinkCallback have to be passed along with the method invocation. In our case, the SDE is PendingMessages and the grid service instance is one of the bulletin service instances. Since the BulletinEditor class implements the NotificationSinkCallback interface, providing the implementation of the deliverNotification() callback method, as shown later, its reference is passed to the addListner() method. It should be noted that addListner() is invoked inside the for loop and is invoked once for each of the bulletin service instances. Further, since the same reference of the NotificationSinkCallback instance ‘this’ is passed in with multiple invocations of the addListner, the deliverNotification() method of this instance will receive notifications from the SDEs of multiple bulletin service instances.

Figure 7-37 on page 169 shows the continuation of the run() method. Once the initialization and registration are performed, the implementation of the run()

method allows the editor to review the pending messages one by one. Since there are multiple topics and hence multiple instances of the bulletin service from which the editor can retrieve messages, the editor can choose the topic of the bulletin messages he/she wants to review and approve. As shown in the figure, the pending messages are retrieved by invoking the **getPendingMessages** operation. The **getPendingMessages** operation retrieves all the messages that were submitted by the writer on the given topic since the last invocation of this operation. The implementation splits the set of messages into individual messages and allows the editor to review the message. Subsequently, when the editor approves the message, the **submitApprovedMessages** operation is invoked on the proper bulletin service instance. This operation stores the approved messages in the ApprovedMessages Service Data Element and pushes the messages to the subscriber.

```
boolean go = true;
          while (go) {
             System.out.println("Type:\n\t R - review pending messages\n\t Q - quit
program");
             try {
                BufferedReader br = new BufferedReader(new InputStreamReader(System.in));
                char key = Character.toLowerCase((char) br.read());
                switch(key) {
                   case 'r':
                      br.readLine();
                      System.out.println("Which type of message do you want to review ?");
                      String name = br.readLine();
                      int instanceIndex = instanceNames.indexOf(name);
                      String messageChunck = bulletins[instanceIndex].getPendingMessages();
                      String[] messages = splitMessages(messageChunck);
                      for(int i=0, j=messages.length; i<j; i++) {
                         System.out.println("Message ("+(i+1)+"/"+j+"):\n"+messages[i]);
                         System.out.println("Do you approve it (y/n) ?");
                         char opt = Character.toLowerCase((char) br.read());
                         if (opt == 'y') {
                            bulletins[instanceIndex].submitApprovedMessages(messages[i]);
                         }
                         br.readLine();
                      }
                   break;
                   case 'q':
                      go = false;
                   break;
                   default:
                      System.out.println("Whazaaa ???");
                }
             } catch (Exception e) {
                System.err.println("An error ocurred while processing a notificaton: "+e);
                e.printStackTrace();
                go = false;
             }
          }
          // Stop listening
          for (int i=0; i<instanceCount; i++) {
             notifManager.removeListener(sinks[i]);
          }
          notifManager.stopListening();
          System.out.println("Not listening anymore!");
       } catch (Exception e) { e.printStackTrace(); }
    }
```

Figure 7-37 Implementation of the BulletinEditor class - continued

As shown in Figure 7-37 on page 169, while exiting, the run() method removes the listeners and stops listening for the notifications. When a listener is added, a handle to the listener is returned by the runtime. It is stored in the sinks array. The listener is removed by invoking the removeListner() method on the NotificationSinkManager and by passing the handle that was obtained while adding the listener. The listening is stopped by invoking the stopListening() method on the NotificationSinkManager.

Figure 7-38 on page 171 shows the continuation of the BulletinEditor class implementation. The first method, namely, splitMessages(), is the private method used from within the run() method for splitting a set of messages into an array of strings. The deliverNotification() method is a callback method and, as explained below, is invoked by the runtime.

Whenever a news bulletin message is submitted by the writer, the server triggers a change notification. The listener detects the notification event and the notification is delivered to the class implementing the NotificationSinkCallback class. As was shown above, in our example the BulletinEditor class itself implements the NotificationSinkCallback interface and is provided the notifications. Figure 7-38 on page 171 shows the implementation of the deliverNotification() callback method.

```
private String[] splitMessages(String mc) {
       String[] result;
       StringTokenizer tokenizer = new StringTokenizer(mc, "\n");
       List messages = new ArrayList();

       while (tokenizer.hasMoreElements()) {
          messages.add(tokenizer.nextElement());
       }

       result = new String[messages.size()];
       Iterator it = messages.iterator();
       int i = 0;
       while(it.hasNext()) {
          result[i++] = (String) it.next();
       }

       return result;
    }

public void deliverNotification(ExtensibilityType any) throws RemoteException {
       ServiceDataValuesType serviceData = AnyHelper.getAsServiceDataValues(any);
       MessageDataType mdt = (MessageDataType) AnyHelper.getAsSingleObject(serviceData,
MessageDataType.class);

       System.out.println("There are new pending messages about "+mdt.getMessage()+".");
    }
)
```

Figure 7-38 Implementation of the BulletinEditor class - continued

As shown in Figure 7-38, when deliverNotification() is called, the service data is passed in the any variable. The method extracts the name of the News Service delivering the notification from the service data and presents it to the editor. This is a pull notification whereby the editor is notified about the existence of the pending messages on a particular topic and the editor, as shown in the run() method implementation, pulls the bulletin messages from the proper bulletin service instance. The static methods of the AnyHelper class provide the utility functions needed to extract the service data and the topic from the service data. The getAsServiceDatavalues() method extracts the service data from the incoming any variable. The getAsSingleObject() obtains the message of the MessageDataType class from the service data. The method is passed the class object of the message along with service data.

7.7 Phase IV: making the News Service robust

Phase one, two, and three concentrated on satisfying the functional requirements specified in the previous chapter. In this phase, we will specifically focus on making the News Service robust. With the implementation, so far, if the server hosting the service instances is down for some reason, all the pending news bulletin messages will be lost. The approved messages are pushed to the subscriber right after the editor approves them. On the other hand, the pending messages submitted by the writer reside in memory until the editor retrieves, reviews and approves them. Hence, if the server is down, all the pending messages will be lost.

In order to make the News Service robust, we will take advantage of the callback methods which are executed at different stages of the service instances life cycle. 5.4, “Life cycle” on page 64 provides a discussion of the life cycle of the grid service instances and the critical points in the life cycle when the various callback methods are executed.

In our example, we have to make certain that the pending messages will survive server shutdowns and be available when the server is back up and running. In order to ensure that the pending messages survive server shutdowns, we will have to persist them when the News Service instances are destroyed from memory and create them when the server starts up and News Service instances are recreated. The pending messages are created and managed in the BulletinPenMesgOprImpl class. The set of figures shown next illustrate the implementation of the BulletinPenMesgOprImpl class for phase four. The additional code for phase four has been highlighted.

```
package com.ibm.itso.grid.gt3.bulletin.server;

import com.ibm.itso.grid.gt3.bulletin.common.stubs.MessageDataType;

import org.globus.ogsa.GridContext;
import org.globus.ogsa.ServiceData;
import org.globus.ogsa.GridServiceBase;
import org.globus.ogsa.GridServiceException;
import org.globus.ogsa.OperationProvider;
import org.globus.ogsa.GridServiceCallback;
import java.rmi.RemoteException;
import java.io.File;
import java.io.FileInputStream;
import java.io.FileOutputStream;
import java.io.IOException;
import java.io.ObjectInputStream;
import java.io.ObjectOutputStream;
import javax.xml.namespace.QName;

public class BulletinPenMesgOprImpl implements OperationProvider, GridServiceCallback {

   private MainSrvImpl myMain;
   private MessageDataType mdt;
   private String instanceName;
   private String pendingMessageChunck = "";
   private ServiceData pendingMessageSDE;

   private static final QName[] operations = new QName[]{new
QName("http://bulletin.gt3.grid.itso.ibm.com/common/stubs", "submitMessageForApproval"),
                          new QName("http://bulletin.gt3.grid.itso.ibm.com/common/stubs",
"getPendingMessages")};
   private GridServiceBase base;
   public void initialize(GridServiceBase base) throws GridServiceException {
      this.base = base;
   }

      public QName[] getOperations() {
      return operations;
   }

   BulletinPenMesgOprImpl(MainSrvImpl main) {
      myMain = main;
   }
```

Figure 7-39 Implementation of the BulletinPenMesgOprImpl Class

```
public void submitMessageForApproval(java.lang.String msg) throws RemoteException {
   System.out.println("Instance "+instanceName+" received message: " +msg);
   pendingMessageChunck += msg + "\n";

   pendingMessageSDE.notifyChange();
}

public java.lang.String getPendingMessages() throws RemoteException {
   String result = new String(pendingMessageChunck);
   pendingMessageChunck = "";
   return result;
}

public void preCreate(GridServiceBase arg0) throws GridServiceException {
   System.out.println("Service instance will be created");
}

public void postCreate(GridContext arg0) throws GridServiceException {
   instanceName = myMain.getInstanceName();
   System.out.println("Instance "+instanceName+" created.");

   pendingMessageSDE = base.getServiceDataSet().create("PendingMessages");

   mdt = new MessageDataType();
   pendingMessageSDE.setValue(mdt);
   mdt.setMessage(instanceName);
   base.getServiceDataSet().add(pendingMessageSDE);
   retrieveMessagesInFile();
}
public void activate(GridContext arg0) throws GridServiceException {
   System.out.println("Service instance has been activated");
}
public void deactivate(GridContext arg0) throws GridServiceException {
   System.out.println("Service instance has been deactivated");
}
public void preDestroy(GridContext arg0) throws GridServiceException {
   saveMessagesInFile();
}
```

Figure 7-40 Implementation of the BulletinPenMesgOprImpl class - continued

```
    //class level lock to avoid different instances access the same file
    private static Object fileLock = new Object();

    //save the messages into data file when your instance will died
    //the file will end with instance name
    private void saveMessagesInFile() {
       synchronized (fileLock) { //necessary only if multi instance with the same name
          File MessageFile = new File("/tmp/data_" + instanceName + ".dat");
          try {
             if (MessageFile.exists())
                MessageFile.delete();
             MessageFile.createNewFile();
             FileOutputStream fileOutS = new FileOutputStream(MessageFile);
             ObjectOutputStream output =
                new ObjectOutputStream(new FileOutputStream(MessageFile));
             output.writeObject(pendingMessageChunck);
             output.flush();
             fileOutS.close();
          } catch (IOException e) {
          }
       }
    }

    //get the messages back from saved file
    private void retrieveMessagesInFile() {
       synchronized (fileLock) { //necessary only if multi instance with the same name
          try {
             File MessageFile = new File("/tmp/data_" + instanceName + ".dat");
             FileInputStream fileInputS = new FileInputStream(MessageFile);
             ObjectInputStream input = new ObjectInputStream(fileInputS);
             pendingMessageChunck = (String) input.readObject();
             fileInputS.close();
             System.out.println(pendingMessageChunck);
          } catch (Exception e) {
          }
       }
    }
}
```

Figure 7-41 Implementation of the BulletinPenMesgOprImpl class - continued

As can be observed from the figures, the postCreate() and preDestroy() callback methods are modified to deal with the persistence of the pending messages. The postCreate() method is invoked by the runtime after the bulletin grid service is created and the preDestroy() callback is invoked by the runtime right before the grid service instance is destroyed. Hence, the pending messages are stored in

permanent storage right before the news grid service is destroyed from memory by invoking the saveMessagesInFile() method from inside the preDestroy(). The pending messages are retrieved from permanent storage and instantiated in memory right after the grid news instance is created and initialized in the postCreate() method. The saveMessagesInFile() method creates a new file if one does not exist and streams the data in the pendingMessageChunk variable, which stores the pending messages. The retrieveMessagesInFile() method opens the file, reads the data and instantiates the pending messages in the pendingMessageChunk variable.

8

IBM Grid Toolbox basics

This chapter introduces the IBM Grid Toolbox V3 for Multiplatforms V1.1, the IBM implementation of the OGSI 1.0 specification.

8.1 Introduction

IBM has been involved in the Globus project (Globus Alliance) for a number of years. After identifying some gaps in the feature set of Globus Toolkit 2.4, a package called the IBM Grid Toolbox was developed and provided on an as-is basis on IBM's AlphaWorks Web site:

http://www.alphaworks.ibm.com

As the Globus project evolved, the IBM Grid Toolbox team determined that due to demand from IBM's enterprise customers, there continued to be a need for a product from IBM that enhanced Globus. In March 2004, the IBM Grid Toolbox V3 for Multiplatforms V1.1 was announced. It provides simple installation and integration of the middleware. The IBM Grid Toolbox also adds significant value for the two major grid roles:

- Grid Developer: tools to develop and test grid services and grid applications
- Grid Administrator: tools to host grid services and grid applications

The IBM Grid Toolbox V3 for Multiplatforms V1.1 implements the OGSI standards and provides the tools to build, develop, deploy, and manage grid services. The IBM Grid Toolbox consists of the following:

- A hosting environment capable of running grid services and collaborating with other grid participants in running large tasks.
- A set of tools to manage, monitor, and administer grid services and the grid hosting environment, including a Web-based interface, called IBM Grid Services Manager.
- A set of APIs and development tools to create and deploy new grid services and grid applications.
- A set of tools to simplify the installation process and the integration of the embedded middleware, such as IBM WebSphere Application Server-Express V5.0.2.

For more information about the IBM Grid Toolbox, refer to the redbook *Grid Computing with the IBM Grid Toolbox*, SG24-6332.

8.1.1 Goals

The IBM Grid Toolbox's primary goal is to provide a common infrastructure for grid computing, autonomic management, and on demand solutions to the IT industry.

Target audience

The target audience for the IBM Grid Toolbox is the enterprise-class developer. IBM has worked with these customers for decades and understands their unique and comprehensive requirements.

The development of grid services and applications is done at a very technically demanding level and requires much effort. The IBM Grid Toolbox is positioned for customers who have already decided to implement a grid, have studied and are comfortable with Globus Toolkit 3, but are looking for a supported, licensed product to match the support model of their other hardware and software choices.

Enterprise value

The IBM Grid Toolbox brings value over and beyond what is available from the open source community:

- Accelerates grid utilization since it provides a more complete development and administration environment than GT3 alone.
- Lowers the risk inherent in developing with GT3 alone, since it is an IBM-supported product (this support must be purchased).
- Allows enterprises to leverage the heterogeneous nature of their IT infrastructure.
- Complements IBM's grid industry offerings. For details, see:

  http://www.ibm.com/grid/solutions/index.shtml
- Installs around the network easily with GUI or automated tools.
- Hosts components and common services within the embedded version of the IBM WebSphere Application Server -Express V5.0.2.
- Scales with minimal incremental overhead.
- Coexists with other instances of WebSphere Application Server and other Web services products.
- Interoperates with other standards-compliant implementations.

8.1.2 Services

The IBM Grid Toolbox includes core grid services and base grid services. Core grid services are always available in the IBM Grid Toolbox instance. They cannot be deployed or undeployed except during the installation process. Core grid services include container management, logging services and security services. Base grid services can be deployed during the IBM Grid Toolbox installation or they can be deployed or undeployed separately. Base grid services include Information services, Data Management services, Program Management

services, Common Management Model Services, Policy services, and Service Group services. The base grid services can be selectively installed.

8.2 Tooling

The IBM Grid Toolbox includes a Software Development Kit (SDK) that provides a collection of reference information and tools for grid service developers. These tools are presented in each methodology phase.

The coding tools themselves are based in the GT3. However, the IBM Grid Toolbox not only encapsulate all these GT3 tools, but also offers complementary tools for deploying, testing and management; this facilitates and improves the whole process of developing grid applications.

8.2.1 Coding and building

The coding and building process can be automated in one single step though the use of the **`ant`** tool, using different types of inputs. These are presented next.

The possible alternatives to start the process are to have as input a Java code (bottom-up approach), a GWSDL code (top-down approach), or an XML code (batch service creation approach).

The output of this process is the JAR and GAR files, meaning the stubs, service locators, deployment descriptors, and an operator provider. Eventually, some application-specific business logic may be added, so the packaging process should be done manually.

Bottom-up approach This approach has as input Java code in a JAR file; this means that the code should already be compiled and archived in a JAR file.
Afterwards, the build.xml script may be customized with any other suitable property, and the **`ant`** command executed.
Remember that, as stated previously, some Java code may have problems in the WSDL mapping, so this automated process may not work properly.

Top-down approach This approach has GWSDL as input. The build.xml script may be customized with any other suitable property, and the **`ant`** command executed.

Batch service approach This approach has as input XML code following the CreateGridServices.xsd, and allows the creation of

multiple grid services, using both the top-down or the bottom-up approach.

8.2.2 Deployment

During the installation wizard process, the ibmgrid user is created. This user is considered to be the grid administrator and is configured with the proper file system permissions to start and stop the container as well as deploy and undeploy grid service archives (gar files).

In order to set the required environment variables, the ibmgrid user must source the **igt-setenv.sh** script located in /opt/IBMGrid. This can be accomplished with the following command, as illustrated in Example 8-1.

Example 8-1 setenv.sh

```
. /opt/IBMGrid/igt-setenv.sh
```

Once the environment has been configured, the grid administrator (ibmgrid user) can deploy and undeploy services. To deploy a service, the grid archive file must be located in the $GLOBUS_LOCATION/gars directory. The grid administrator will change directories to $GLOBUS_HOME and execute the **igt-deploy-gar** command. Example 8-2 shows how to deploy the servicegroup.gar.

Example 8-2 Deploy services

```
igt-deploy-gar gars/servicegroup.gar
```

To undeploy a gar, the grid administrator (ibmgrid) will execute the **igt-undeploy-gar** command and specify the gar_id as an argument. The gar_id is the name of the grid archive minus the ".gar" extension. For example, the gar_id of the service deployed in Example 8-2 is *ServiceGroup*. Refer to Example 8-3, which illustrates how to undeploy the *ServiceGroup* service.

Example 8-3 Undeploy services

```
igt-undeploy-gar servicegroup
```

8.2.3 Testing

Once a grid service has been deployed into a container, there are ways to test the new grid.

The first one is to use the IBM Grid Services Manager. This tool provides a great deal of useful information about the service, Port Types, WSDL, and operations,

and allows you to manage its status and edit instances properties through a graphical Web interface shown in Figure 8-1.

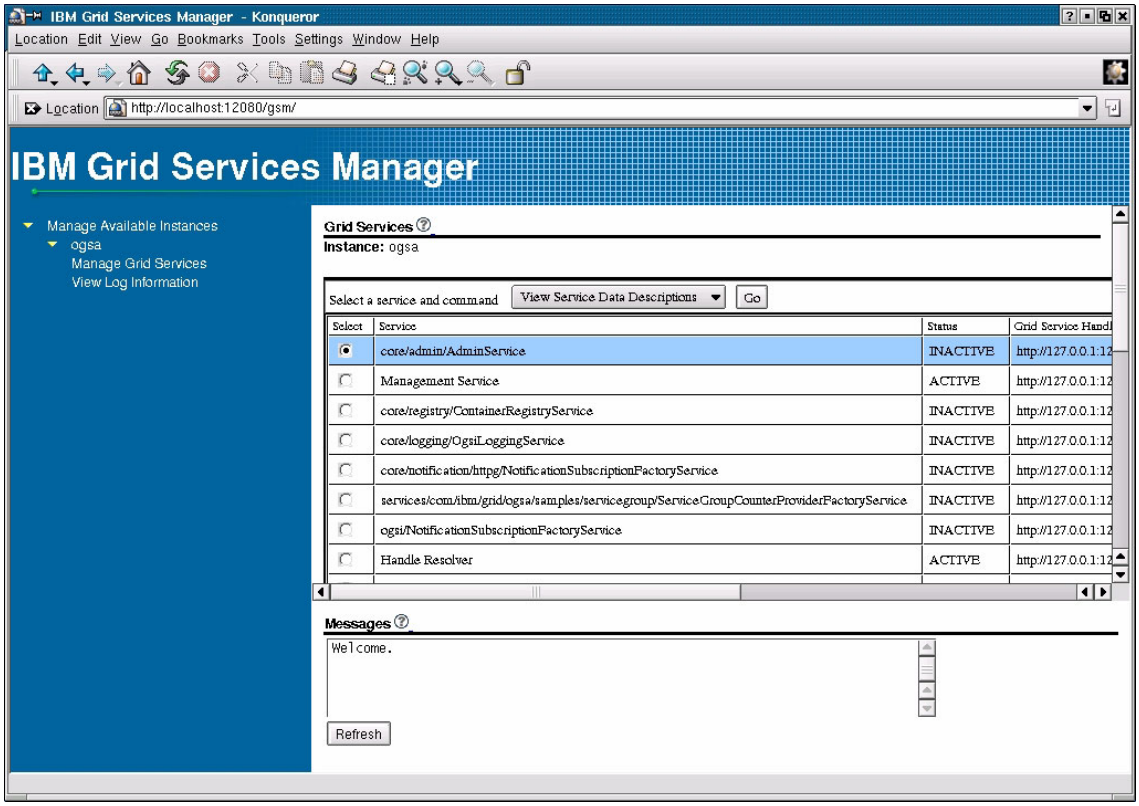


Figure 8-1 IBM Grid Service Manager interface

Some quick interaction may take place using the Service Browser tool, detailed in Appendix F, “Service Browser” on page 253 and also available in the IBM Grid Toolbox. Through the use of an Web interface, it will allow you to test remote methods invocation from a runtime generated form. Watch for the address changes from GT3 to IBM Grid Toolbok.

Finally, the most comprehensive way is to implement a simple client that tests each of its exported methods in a more specific fashion. Such a client should obtain a reference to a service factory, create its own instance and issue calls to each of its methods in a convenient way.

8.3 Case study

The purpose of the case study is to identify and document the differences between the GT3 and IBM Grid Toolbox from developers' and deployers' perspectives.

This document has introduced a simple grid service, then enhanced that service to demonstrate advanced features as they were introduced throughout the text. The sample grid service architecture, design, and development were explained, as seen in Chapter 7, "Case study: grid application enablement" on page 115. The sample code listed in Appendix A, "Sample code" on page 191 that was previously developed using GT3 could have easily been developed using the IBM Grid Toolbox.

The approach used to gather this information was first to redeploy the grid service on the IBM Grid Toolbox and verify the proper operation of the service using GT3 clients, and second, to develop the clients using the IBM Grid Toolbox and verify interoperability with a GT3-hosted grid service. After verifying interoperability of the IBM Grid Toolbox and GT3 environments, all phases of the development effort which were completed on GT3 could be duplicated using the IBM Grid Toolbox.

This general approach was further divided into phases, with each phase incrementally building on the previous one.

1. Deploy a GT3 developed grid service on an IBM Grid Toolbox node using the GT3 deployment process and verify proper operation using the existing GT3 BulletinAdminConsole.
2. Deploy an updated GT3 developed grid service on an IBM Grid Toolbox node using the **`igt-deply-gar`** scripts and verify proper operation by using multiple unique clients to check the full functionality of the service.
3. Build the entire service on the IBM Grid Toolbox from the Java interface specification and implementation.
4. Build clients on the IBM Grid Toolbox which interacted with the existing GT3 grid service.

Prerequisites

IBM Grid Toolbox was installed on Red Hat Advanced Server V2.1. As part of the environment setup and configuration, Apache **`ant`** was installed in /usr/local/apache-ant-1.5.4 and /usr/local/apache-ant-1.5.4/bin was appended to the PATH environment variable. The JAVA_HOME environment variable was set to `/usr/IBMJava2-131`.

Note: The IBM Grid Toolbox is installed and configured using an installation wizard, which greatly simplifies the installation and configuration process.

8.3.1 Case study - phase I

The initial testing phase consisted of a portability test to verify that a grid archive (gar file) exported from the Globus Toolkit could be imported into the IBM Grid Toolbox and run without modifications.

The server-side code is exported as Bulletin.gar and copied to the same directory structure on a machine where the IBM Grid Toolbox was installed and configured.

The environment is set up by sourcing the **igt-setenv.sh** shell script from the /opt/IBMGrid directory; this does not happen with GT3.

The Bulletin.gar file is then deployed using the same **ant** procedure as was used for GT3.

The script shown in Example 8-4 initializes some environment variables by calling the Globus setenv.sh shell script for GT3. Although IBM Grid Toolbox provides the **igt-deploy-gar** command, developers would probably use the **ant** command to deploy and build the grid services.

Example 8-4 IBM Grid Toolbox environment setup and using ant to deploy a gar

```
. /opt/IBMGrid/igt-setenv.sh
ant deploy -Dgar.name=/home/itso/phase3/servicebuild/services/bulletin/\
build/gar/Bulletin.gar
```

Important: The **igt-setenv.sh** wraps the GT3 **setenv** and the **globus-user-env** scripts and also sets additional environment variables for WebSphere Application Server, IBM Grid Toolbox, CloudScape Database, Path, and ClassPath.

Next, a proxy is created for the ibmgrid user and the container is started under that user ID, as shown in Example 8-5. The commands and procedure for creating a proxy are the same for both platforms.

Example 8-5 Create a proxy and start the container

```
grid-proxy-init
igt-start-container
```

Important: The difference lies in the commands to start and stop the container. These are unique in each environment. To start or stop the embedded WebSphere Application Server container for IBM Grid Toolbox, use **`igt-start-container`** or **`igt-stop-container`**.

The Grid Services Manager (GSM) is used to verify the deployment by ensuring that BulletinFactory appears in the services list.

Note: GSM is a GT3 System Level service which is packaged and deployed with the IBM Grid Toolbox. It provides a Web-based user interface for grid administrators to manage instances and services. It allows the grid administrator to add, remove, activate or deactivate instances and edit their properties. It displays the service data descriptions, service port types, or the service WSDL, and provides tools for viewing logging information, as well as adding and removing logging services.

BulletinFactory is selected from the list of services by clicking the corresponding radio button, selecting **Activate** from the GSM drop-down command list, and finally clicking **Go** to activate the service.

Important: This is another difference between GT3 and IBM Grid Toolbox. GT3 uses the service browser which has limited functionality and is not intuitively obvious to use. The IBM Grid Toolbox hosts a Grid Services Manager which has more functionality and is easier to navigate and use than the service browser.

The GT3 BulletinAdminConsole client is used to verify proper operation of the newly deployed IBM Grid Toolbox bulletin service. After proper operation is verified, the container is stopped (using **`igt-stop-container`**) and the bulletin service is undeployed using the same **`ant`** commands and build.xml file that were used in GT3, as shown in Example 8-6.

Example 8-6 Stop the container and undeploy the bulletin service

```
igt-stop-container
Ant undeploy -Dgar.id=Bulletin
```

Important: The URL argument for the clients does need to change to reflect the <server_name> of the IBM Grid Toolbox server and the port on which the IBM Grid Toolbox embedded WebSphere Application Server container is listening. The default port for the IBM Grid Toolbox is 12080, as opposed to the default port for Axis, which is 8080.

The server name is something that will obviously change, but the port of the container is something that also needs to be considered, especially in a heterogeneous environment with GT3 and IBM Grid Toolbox nodes. The IBM Grid Toolbox provides a script that automates changing the port that the container listens on, so the process is easy. Changing the default port for the Globus Toolkit container is a manual process.

Important: Another difference between the GT3 and IBM Grid Toolbox is the port on which the container listens for connections. The IBM Grid Toolbox automates the process of changing the default port number, whereas it is a manual process for the GT3.

8.3.2 Case study - phase II

The grid service used for this phase included additional functionality. The additional functionality was verified using the BulletinWriter and BulletinSubscriber clients.

The updated grid service is deployed using the IBM Grid Toolbox deploy and undeploy scripts in place of the **ant** script which was verified in the previous phase. The gar is deployed using the **igt-deploy-gar** command, as shown in Example 8-7.

Example 8-7 Deploying a gar file using the IBM Grid Toolbox commands

```
igt-deploy-gar
/home/itso/phase4/servicebuild/services/bulletin/build/gar/Bulletin.gar
```

Important: The **igt-deploy-gar**, **igt-undeploy-gar**, and **igt-undeploy-all** commands are provided with the IBM Grid Toolbox and assist the grid deployer in administering the grid environment.

The service is successfully deployed using **igt-deploy-gar** and activated using the IBM Grid Toolbox Grid Services Manager. The GT3 clients are used to verify proper operation of the IBM Grid Toolbox deployed grid service.

In addition to the URL argument changes mentioned for the previous phase, the schema path (to the ogsi_notification_service_sink.wsdl and other associated WSDL files) need to be updated. This is necessary for the new clients that are introduced in this phase. The new clients are the BulletinSubscriber and the BulletinWriter.

The BulletinWriter client submits content to an instance of the BulletinService, for example *news*. The BulletinSubscriber subscribes to an instance, then receives updates when new content is posted to that instance.

Both of these clients are invoked from the Java command line with a parameter specifying the URL of the service and setting a system property for the schema root. Since the clients have been designed and developed to be invoked with command line arguments, changing the argument(s) is the most straightforward approach. If this has not been specified using command line arguments, a possible alternative approach may be to identify the new schema root by editing the value of the schemaPath parameter in the Web service deployment descriptor (WSDD) file.

Important: In GT3, the schema path is off the $GLOBUS_LOCATION which was /usr/local/globus for the redbook development environment. In the IBM Grid Toolbox environment, the schema path is located off $GLOBUS_LOCATION/AppServer/installedApps/IBMGrid.ear/ogsa.war, which is simply mapped to *ogsa*.

Example 8-8 shows the command for running the BulletinSubscriber client. Similarly, Example 8-9 shows the command for running the BulletinSubscriber client on the IBM Grid Toolbox.

Example 8-8 Running the BulletinSubscriber client on GT3

```
Java -Dorg.globus.ogsa.schema.root=http://j2:12080/
com.ibm.itso.grid.gt3.bulletin.client.BulletinSubscriber
http:// j2:12080/ogsa/services/Bulletin/BulletinFactory
```

The **-D** argument provided on the Java command line sets a system property. In this case, the property is org.globus.ogsa.schema.root. This property identifies the location of important WSDL files and is set to the root directory on GT3 and to the OGSA directory on the IBM Grid Toolbox.

Important: Note that the first argument which specifies `org.globus.ogsa.schema.root` is appended with `ogsa`, which allows the IBM Grid Toolbox container to find the WSDL file located in the schema directory at runtime.

Example 8-9 Running the BulletinSubscriber client on IBM Grid Toolbox

```
Java -Dorg.globus.ogsa.schema.root=http://j2:12080/ogsa/
com.ibm.itso.grid.gt3.bulletin.client.BulletinSubscriber
http:// j2:12080/ogsa/services/Bulletin/BulletinFactory
```

8.3.3 Case study - phase III

Up to this point, grid services developed on GT3 have been deployed to the IBM Grid Toolbox and verified using the IBM Grid Toolbox Grid Services Manager and GT3 clients to verify the proper operation of the IBM Grid Toolbox grid service. This has verified interoperability as well as administration from the deployer's perspective.

Now the focus changes to the developer perspective. The Java interface and implementation source code are used to build the grid service in the IBM Grid Toolbox environment.

First, it is important to source the **`igt-setenv.sh`** script located in /opt/IBMGrid or $GLOBUS_LOCATION to set up up the proper environment variables.

As mentioned previously, the default port the IBM Grid Toolbox container listens on is different from the default port of the Axis server. If the source tree from the GT3 environment is duplicated on the IBM Grid Toolbox machine, the ogsa.properties file will need to be updated to correctly reflect the service.port.

Important: The ogsa.properties file contains a service.port directive which needs to be updated to reflect the service.port of the IBM Grid Toolbox 12080 (`service.port=12080`).

When building the source on GT3, the outcome does not depend on the status of the container. The container can be running or stopped during the build process. However, with IBM Grid Toolbox, the ogsa.properties file must reflect the correct service.port and the container must be running for the **`ant`** build.xml script to complete successfully.

With these changes, the updated Bulletin service is successfully built from source using the IBM Grid Toolbox. The service is packaged and deployed, then verified using the existing GT3 clients (BulletinAdminConsole, BulletinWriter, BulletinSubscriber, and BulletinEditor). Note that the command line arguments for the clients must reflect `ogsa` for the schema root, as illustrated in Example 8-9 on page 187.

8.3.4 Case study - phase IV

The **`ant`** build.xml file that was generated for GT3 development has already been verified to work with the IBM Grid Toolbox. Placing the client source code in the appropriate directory structure and running the **`ant`** script will build the clients since the clients were designed and developed to accept command line arguments. The clients can be built and used in the same manner on GT3 or on the IBM Grid Toolbox.

Example 8-10 is for an IBM Grid Toolbox client that will use an existing GT3 grid service. The only difference is in the server name of either the GT3 machine or the IBM Grid Toolbox machine. However, when the clients connect to an IBM Grid Toolbox node, the parameter for org.globus.ogsa.schema.root needs to reflect the `ogsa` directory.

Example 8-10 BulletinSubscriber client using a GT3 hosted BulletinFactory service

```
Java -Dorg.globus.ogsa.schema.root=http://i2:8080/
com.ibm.itso.grid.gt3.bulletin.client.BulletinSubscriber
http:// i2:8080/ogsa/services/Bulletin/BulletinFactory
```

Please see the following example of a client accessing an IBM Grid Toolbox grid service. Notice that the server name, server port, and URI of the schema root have been updated to `j2:12080/ogsa/`.

Example 8-11 BulletinSubscriber client using IBM Grid Toolbox hosted BulletinFactory service

```
Java -Dorg.globus.ogsa.schema.root=http://j2:12080/ogsa/
com.ibm.itso.grid.gt3.bulletin.client.BulletinSubscriber
http:// j2:12080/ogsa/services/Bulletin/BulletinFactory
```

At this point in the case study, all issues have been identified and resolved. Clients are able to interact with services in either a GT3 container or an IBM Grid Toolbox container. GT3 and IBM Grid Toolbox clients can administer, subscribe to, write or approve content for either environment. This verifies the interoperability of GT3 and IBM Grid Toolbox and demonstrates how the platforms can co-exist in a heterogeneous environment. This case study has demonstrated that code can be developed on one platform and deployed to another. The concepts presented here are specific to the bulletin service example but can be applied to any grid service.

A

Sample code

This appendix contains part of the source codes of the bulletin service developed for this project. This includes the server and the client sides, the Java code, and the GWSDL schema files.

Server-side code

This session contains the server-side source code.

Example: A-1 GWSDL schema file for bulletin example

```
<?xml version="1.0" encoding="UTF-8"?>
<definitions name="BulletinService"
    targetNamespace="http://bulletin.gt3.grid.itso.ibm.com/common/stubs"
    xmlns:tns="http://bulletin.gt3.grid.itso.ibm.com/common/stubs"
    xmlns:data="http://bulletin.gt3.grid.itso.ibm.com/common/stubs"
    xmlns:ogsi="http://www.gridforum.org/namespaces/2003/03/OGSI"
    xmlns:gwsdl="http://www.gridforum.org/namespaces/2003/03/gridWSDLExtensions"
    xmlns:sd="http://www.gridforum.org/namespaces/2003/03/serviceData"
    xmlns:xsd="http://www.w3.org/2001/XMLSchema"
    xmlns="http://schemas.xmlsoap.org/wsdl/">

<import location="../../ogsi/ogsi.gwsdl"
namespace="http://www.gridforum.org/namespaces/2003/03/OGSI"/>

<import location="MessageDataType.xsd"
namespace="http://bulletin.gt3.grid.itso.ibm.com/common/stubs"/>



<types>
<xsd:schema targetNamespace="http://bulletin.gt3.grid.itso.ibm.com/common/stubs"
        attributeFormDefault="qualified"
        elementFormDefault="qualified"
   xmlns="http://www.w3.org/2001/XMLSchema">
  <xsd:element name="submitMessageForApproval">
    <xsd:complexType>
      <xsd:sequence>
        <xsd:element name="value" type="xsd:string"/>
      </xsd:sequence>
    </xsd:complexType>
  </xsd:element>
  <xsd:element name="submitMessageForApprovalResponse">
    <xsd:complexType/>
  </xsd:element>
  <xsd:element name="getPendingMessages">
    <xsd:complexType/>
  </xsd:element>
  <xsd:element name="getPendingMessagesResponse">
    <xsd:complexType>
      <xsd:sequence>
        <xsd:element name="value" type="xsd:string"/>
      </xsd:sequence>
    </xsd:complexType>
```

```
  </xsd:element>
  <xsd:element name="submitApprovedMessages">
    <xsd:complexType>
      <xsd:sequence>
        <xsd:element name="value" type="xsd:string"/>
      </xsd:sequence>
    </xsd:complexType>
  </xsd:element>
  <xsd:element name="submitApprovedMessagesResponse">
    <xsd:complexType/>
  </xsd:element>
</xsd:schema>
</types>

<message name="SubmitForApprovalInputMessage">
  <part name="parameters" element="tns:submitMessageForApproval"/>
</message>
<message name="SubmitForApprovalOutputMessage">
  <part name="parameters" element="tns:submitMessageForApprovalResponse"/>
</message>
<message name="GetPendingInputMessage">
  <part name="parameters" element="tns:getPendingMessages"/>
</message>
<message name="GetPendingOutputMessage">
  <part name="parameters" element="tns:getPendingMessagesResponse"/>
</message>
<message name="SubmitApprovedInputMessage">
  <part name="parameters" element="tns:submitApprovedMessages"/>
</message>
<message name="SubmitApprovedOutputMessage">
  <part name="parameters" element="tns:submitApprovedMessagesResponse"/>
</message>

<gwsdl:portType name="BulletinPortType" extends="ogsi:GridService ogsi:NotificationSource">
  <operation name="submitMessageForApproval">
    <input message="tns:SubmitForApprovalInputMessage"/>
    <output message="tns:SubmitForApprovalOutputMessage"/>
    <fault name="Fault" message="ogsi:FaultMessage"/>
  </operation>
  <operation name="getPendingMessages">
    <input message="tns:GetPendingInputMessage"/>
    <output message="tns:GetPendingOutputMessage"/>
    <fault name="Fault" message="ogsi:FaultMessage"/>
  </operation>
  <operation name="submitApprovedMessages">
    <input message="tns:SubmitApprovedInputMessage"/>
    <output message="tns:SubmitApprovedOutputMessage"/>
    <fault name="Fault" message="ogsi:FaultMessage"/>
  </operation>
```

```
  <sd:serviceData name="PendingMessages"
                  type="data:MessageDataType"
                  minOccurs="1"
                  maxOccurs="1"
                  mutability="mutable"
                  modifiable="false"
                  nillable="false">
  </sd:serviceData>
  <sd:serviceData name="ApprovedMessages"
                  type="data:MessageDataType"
                  minOccurs="1"
                  maxOccurs="1"
                  mutability="mutable"
                  modifiable="false"
                  nillable="false">
  </sd:serviceData>
</gwsdl:portType>

</definitions>
```

Example: A-2 MessageDataType.xsd (needed in the same directory as Bulletin.gwsdl)

```
<?xml version="1.0" encoding="UTF-8"?>
<wsdl:definitions name="MessageData"
   targetNamespace="http://bulletin.gt3.grid.itso.ibm.com/common/stubs"
   xmlns:tns="http://bulletin.gt3.grid.itso.ibm.com/common/stubs"
   xmlns:wsdl="http://schemas.xmlsoap.org/wsdl/">

<wsdl:types>
<schema targetNamespace="http://bulletin.gt3.grid.itso.ibm.com/common/stubs"
   attributeFormDefault="qualified"
   elementFormDefault="qualified"
   xmlns="http://www.w3.org/2001/XMLSchema">

   <complexType name="MessageDataType">
      <sequence>
         <element name="message" type="string"/>
      </sequence>
   </complexType>

</schema>
</wsdl:types>

</wsdl:definitions>
```

Example: A-3 Code for BulletinPenMesgOprImpl.java

```
package com.ibm.itso.grid.gt3.bulletin.server;

import com.ibm.itso.grid.gt3.bulletin.common.stubs.MessageDataType;

import org.globus.ogsa.GridContext;
import org.globus.ogsa.ServiceData;
import org.globus.ogsa.GridServiceBase;
import org.globus.ogsa.GridServiceException;
import org.globus.ogsa.OperationProvider;
import org.globus.ogsa.GridServiceCallback;

import java.rmi.RemoteException;

import java.io.File;
import java.io.FileInputStream;
import java.io.FileOutputStream;
import java.io.IOException;
import java.io.ObjectInputStream;
import java.io.ObjectOutputStream;
import javax.xml.namespace.QName;


public class BulletinPenMesgOprImpl implements OperationProvider, GridServiceCallback {

        private MainSrvImpl myMain;

        private MessageDataType mdt;

        private String instanceName;
        private String pendingMessageChunck = "";

        private ServiceData pendingMessageSDE;

        // Operation provider properties
        private static final QName[] operations = new QName[]{new
QName("http://bulletin.gt3.grid.itso.ibm.com/common/stubs", "submitMessageForApproval"),
                                                              new
QName("http://bulletin.gt3.grid.itso.ibm.com/common/stubs", "getPendingMessages")};
        private GridServiceBase base;

        /* (non-Javadoc)
         * @see org.globus.ogsa.OperationProvider#initialize(org.globus.ogsa.GridServiceBase)
         */
        public void initialize(GridServiceBase base) throws GridServiceException {
                this.base = base;
        }
```

```
/* (non-Javadoc)
 * @see org.globus.ogsa.OperationProvider#getOperations()
 */
public QName[] getOperations() {
        return operations;
}

BulletinPenMesgOprImpl(MainSrvImpl main) {
        myMain = main;
}

/**
 * This method should be invoked by writers when they want to submit
 * a new message. It fires a notification to the editor
 * informing him about the new pending messages waiting for his
 * approval.
 */
public void submitMessageForApproval(java.lang.String msg) throws RemoteException {
        System.out.println("Instance "+instanceName+" received message: " +msg);
        pendingMessageChunck += msg + "\n";

        pendingMessageSDE.notifyChange();
}

/**
 * This method provides the editor with the messages he hasn't taken
 * so far.
 */
public java.lang.String getPendingMessages() throws RemoteException {
        String result = new String(pendingMessageChunck);
        pendingMessageChunck = "";
        return result;
}

/* (non-Javadoc)
 * @see org.globus.ogsa.GridServiceCallback#preCreate(org.globus.ogsa.GridServiceBase)
 */
public void preCreate(GridServiceBase arg0) throws GridServiceException {
        System.out.println("Service instance will be created");
}

/* (non-Javadoc)
 * @see org.globus.ogsa.GridServiceCallback#postCreate(org.globus.ogsa.GridContext)
 */
public void postCreate(GridContext arg0) throws GridServiceException {
        instanceName = myMain.getInstanceName();
        System.out.println("Instance "+instanceName+" created.");

        pendingMessageSDE = base.getServiceDataSet().create("PendingMessages");
```

```
        mdt = new MessageDataType();
        pendingMessageSDE.setValue(mdt);
        mdt.setMessage(instanceName);

        base.getServiceDataSet().add(pendingMessageSDE);

        retrieveMessagesInFile();
}

/* (non-Javadoc)
 * @see org.globus.ogsa.GridServiceCallback#activate(org.globus.ogsa.GridContext)
 */
public void activate(GridContext arg0) throws GridServiceException {
        System.out.println("Service instance has been activated");
}

/* (non-Javadoc)
 * @see org.globus.ogsa.GridServiceCallback#deactivate(org.globus.ogsa.GridContext)
 */
public void deactivate(GridContext arg0) throws GridServiceException {
        System.out.println("Service instance has been deactivated");
}

/* (non-Javadoc)
 * @see org.globus.ogsa.GridServiceCallback#preDestroy(org.globus.ogsa.GridContext)
 */
public void preDestroy(GridContext arg0) throws GridServiceException {
        saveMessagesInFile();
}

// *********************************************************
// ***** This is the code that deals with file storage *****
// *********************************************************

//class level lock to avoid different instances access the same file
private static Object fileLock = new Object();

//save the messages into data file when your instance will died
//the file will end with instance name
private void saveMessagesInFile() {
        synchronized (fileLock) { //necessary only if multi instance with the same name
                File MessageFile = new File("/tmp/data_" + instanceName + ".dat");
                try {
                        if (MessageFile.exists())
                                MessageFile.delete();
                        MessageFile.createNewFile();
                        FileOutputStream fileOutS = new FileOutputStream(MessageFile);
                        ObjectOutputStream output =
```

```
                                        new ObjectOutputStream(new
FileOutputStream(MessageFile));
                                output.writeObject(pendingMessageChunck);
                                output.flush();
                                fileOutS.close();
                        } catch (IOException e) {
                        }
                }
        }

        //get the messages back from saved file
        private void retrieveMessagesInFile() {
                synchronized (fileLock) { //necessary only if multi instance with the same name
                        try {
                                File MessageFile = new File("/tmp/data_" + instanceName +
".dat");
                                FileInputStream fileInputS = new FileInputStream(MessageFile);
                                ObjectInputStream input = new ObjectInputStream(fileInputS);
                                pendingMessageChunck = (String) input.readObject();
                                fileInputS.close();
                                System.out.println(pendingMessageChunck);
                        } catch (Exception e) {
                        }
                }
        }
}
```

Example: A-4 Code for BulletinAppMesgOprImpl.java

```
package com.ibm.itso.grid.gt3.bulletin.server;

import com.ibm.itso.grid.gt3.bulletin.common.stubs.MessageDataType;

import org.globus.ogsa.GridContext;
import org.globus.ogsa.GridServiceCallback;
import org.globus.ogsa.ServiceData;
import org.globus.ogsa.GridServiceBase;
import org.globus.ogsa.GridServiceException;
import org.globus.ogsa.OperationProvider;

import java.rmi.RemoteException;

import javax.xml.namespace.QName;

public class BulletinAppMesgOprImpl implements OperationProvider, GridServiceCallback {

        private MainSrvImpl myMain;
```

```
        private MessageDataType mdt;

        private String instanceName;

        private ServiceData approvedMessagesSDE;

        // Operation provider properties
        private static final QName[] operations = new QName[]{new
QName("http://bulletin.gt3.grid.itso.ibm.com/common/stubs", "submitApprovedMessages")};
        private GridServiceBase base;

        /* (non-Javadoc)
         * @see org.globus.ogsa.OperationProvider#initialize(org.globus.ogsa.GridServiceBase)
         */
        public void initialize(GridServiceBase base) throws GridServiceException {
                this.base = base;
        }

        /* (non-Javadoc)
         * @see org.globus.ogsa.OperationProvider#getOperations()
         */
        public QName[] getOperations() {
                return operations;
        }

        BulletinAppMesgOprImpl(MainSrvImpl main) {
                myMain = main;
        }

        /**
         * This method is invoked by the editor so that the subscribers can be
         * notified of the messages he has approved.
         */
        public void submitApprovedMessages(java.lang.String msgs) throws RemoteException {
                mdt.setMessage(instanceName+": "+msgs);
                approvedMessagesSDE.notifyChange();
        }

        /* (non-Javadoc)
         * @see org.globus.ogsa.GridServiceCallback#preCreate(org.globus.ogsa.GridServiceBase)
         */
        public void preCreate(GridServiceBase arg0) throws GridServiceException {
                // Do nothing
        }

        /* (non-Javadoc)
         * @see org.globus.ogsa.GridServiceCallback#postCreate(org.globus.ogsa.GridContext)
         */
```

```
        public void postCreate(GridContext arg0) throws GridServiceException {
                instanceName = myMain.getInstanceName();
                System.out.println("Instance "+instanceName+" created.");

                approvedMessagesSDE = base.getServiceDataSet().create("ApprovedMessages");

                mdt = new MessageDataType();
                approvedMessagesSDE.setValue(mdt);
                mdt.setMessage("Initialized");

                base.getServiceDataSet().add(approvedMessagesSDE);
        }

        /* (non-Javadoc)
         * @see org.globus.ogsa.GridServiceCallback#activate(org.globus.ogsa.GridContext)
         */
        public void activate(GridContext arg0) throws GridServiceException {
                // Do nothing
        }

        /* (non-Javadoc)
         * @see org.globus.ogsa.GridServiceCallback#deactivate(org.globus.ogsa.GridContext)
         */
        public void deactivate(GridContext arg0) throws GridServiceException {
                // Do nothing
        }

        /* (non-Javadoc)
         * @see org.globus.ogsa.GridServiceCallback#preDestroy(org.globus.ogsa.GridContext)
         */
        public void preDestroy(GridContext arg0) throws GridServiceException {
                // Do nothing
        }
}
```

Example: A-5 Code for MainSrvImpl.java

```
package com.ibm.itso.grid.gt3.bulletin.server;

import org.globus.ogsa.GridServiceException;
import org.globus.ogsa.impl.ogsi.GridServiceImpl;
import org.gridforum.ogsi.LocatorType;
import javax.xml.namespace.QName;
import org.globus.ogsa.ServiceData;
import org.globus.ogsa.ServiceDataSet;
```

```
public class MainSrvImpl extends GridServiceImpl {

        public MainSrvImpl() throws GridServiceException {
                super("Bulletin Service Implementation");
                                this.addOperationProvider(new BulletinAppMesgOprImpl(this));
                this.addOperationProvider(new BulletinPenMesgOprImpl(this));
        }

        /**
        //get the grid server instance name
        //first get all the GSHs
        //then get all the GSRs
        //Compare them to return the first name we found
        **/
        public String getInstanceName() {

                String Space = "http://www.gridforum.org/namespaces/2003/03/OGSI";
                try {
                        ServiceDataSet dataset = this.getServiceDataSet();
                        QName handle = new QName(Space,"gridServiceHandle");
                        QName factoryLocator = new QName(Space,"factoryLocator");
                        ServiceData dataOfHandle = dataset.get(handle);
                        ServiceData dataOfFactor = dataset.get(factoryLocator);

                        String handleS = dataOfHandle.getValue().toString();

                        LocatorType locatortype = (LocatorType)dataOfFactor.getValue();
                        String factorS= (locatortype.getHandle()[0]).getValue().toString();

                        int lastPosition = handleS.lastIndexOf("/");
                        if (factorS.equalsIgnoreCase(handleS.substring(0,lastPosition  ))) {
                                String InstanceName = handleS.substring(lastPosition +1,
handleS.length());
                                System.out.println("Get instance Name:"+InstanceName);
                                return InstanceName;
                        }
                } catch (Exception e) {
                        e.printStackTrace();
                        return null;
                }
                return null;
        }


}
```

Client-side code

This session contains the client-side source code.

Example: A-6 Code for BulletinEditor.java

```
package com.ibm.itso.grid.gt3.bulletin.client;

import com.ibm.itso.grid.gt3.bulletin.common.stubs.service.BulletinServiceGridLocator;
import com.ibm.itso.grid.gt3.bulletin.common.stubs.BulletinPortType;
import com.ibm.itso.grid.gt3.bulletin.common.stubs.MessageDataType;

import org.globus.ogsa.utils.AnyHelper;
import org.globus.ogsa.NotificationSinkCallback;
import org.globus.ogsa.impl.core.service.ServicePropertiesImpl;
import org.globus.ogsa.client.managers.NotificationSinkManager;

import org.gridforum.ogsi.HandleType;
import org.gridforum.ogsi.ExtensibilityType;
import org.gridforum.ogsi.ServiceDataValuesType;

import java.io.BufferedReader;
import java.io.InputStreamReader;
import java.net.URL;
import java.rmi.RemoteException;
import java.util.List;
import java.util.Iterator;
import java.util.ArrayList;
import java.util.StringTokenizer;

public class BulletinEditor extends ServicePropertiesImpl implements NotificationSinkCallback {

        List instanceNames;
        BulletinPortType[] bulletins;

        public static final void main(String[] args) {
                BulletinEditor be = new BulletinEditor();
                be.run(args);
        }

        private void run(String[] args) {

                // The base GSH and the instance names will be informed
                // as command-line arguments
                String baseGSH = args[0];

                // Build service instance references
                int instanceCount = args.length-1;
                BulletinPortType[] bulletins = new BulletinPortType[instanceCount];
```

```
                instanceNames = new ArrayList(instanceCount);

                try {
                        NotificationSinkManager notifManager =
NotificationSinkManager.getManager();
                        String sinks[] = new String[instanceCount];

                        for (int i=0; i<instanceCount; i++) {
                                String GSHstr = baseGSH+"/"+args[1+i];
                                URL GSH = new URL(GSHstr);
                                BulletinServiceGridLocator bulletinGL = new
BulletinServiceGridLocator();
                                bulletins[i] = bulletinGL.getBulletinServicePort(GSH);


notifManager.startListening(NotificationSinkManager.MAIN_THREAD);
                                sinks[i] = notifManager.addListener("PendingMessages", null,
new HandleType(GSHstr), this);
                                instanceNames.add(args[i+1]);
                        };

                        boolean go = true;
                        while (go) {

                               System.out.println("Type:\n\t R - review pending messages\n\t Q
- quit program");

                                try {
                                        BufferedReader br
                                                        = new BufferedReader(new
InputStreamReader(System.in));

                                        char key = Character.toLowerCase((char) br.read());

                                        switch(key) {
                                                case 'r':
                                                        br.readLine();
                                                        System.out.println("Which type of
message do you want to review ?");
                                                        String name = br.readLine();
                                                        int instanceIndex =
instanceNames.indexOf(name);
                                                        String messageChunck =
bulletins[instanceIndex].getPendingMessages();
                                                        String[] messages =
splitMessages(messageChunck);
                                                        for(int i=0, j=messages.length; i<j;
i++) {
```

```
                                                                System.out.println("Message
("+(i+1)+"/"+j+"):\n"+messages[i]);
                                                                System.out.println("Do you
approve it (y/n) ?");
                                                                char opt =
Character.toLowerCase((char) br.read());
                                                                if (opt == 'y') {

bulletins[instanceIndex].submitApprovedMessages(messages[i]);
                                                                }
                                                                br.readLine();
                                                        }
                                                break;
                                                case 'q':
                                                        go = false;
                                                break;
                                                default:
                                                        System.out.println("Whazaaa ???");
                                        }
                                } catch (Exception e) {
                                       System.err.println("An error ocurred while processing a
notificaton: "+e);
                                        e.printStackTrace();
                                        go = false;
                                }
                        }

                        // Stop listening
                        for (int i=0; i<instanceCount; i++) {
                                notifManager.removeListener(sinks[i]);
                        }

                        notifManager.stopListening();
                        System.out.println("Not listening anymore!");
                } catch (Exception e) {
                        e.printStackTrace();
                }
        }

        /* (non-Javadoc)
         * @see
org.gridforum.ogsi.NotificationSink#deliverNotification(org.gridforum.ogsi.ExtensibilityType)
         */
        public void deliverNotification(ExtensibilityType any) throws RemoteException {
                ServiceDataValuesType serviceData = AnyHelper.getAsServiceDataValues(any);
                MessageDataType mdt = (MessageDataType)
AnyHelper.getAsSingleObject(serviceData, MessageDataType.class);
```

```
                System.out.println("There are new pending messages about
"+mdt.getMessage()+".");
        }

        private String[] splitMessages(String mc) {
                String[] result;
                StringTokenizer tokenizer = new StringTokenizer(mc, "\n");
                List messages = new ArrayList();

                while (tokenizer.hasMoreElements()) {
                        messages.add(tokenizer.nextElement());
                }

                result = new String[messages.size()];
                Iterator it = messages.iterator();
                int i = 0;
                while(it.hasNext()) {
                        result[i++] = (String) it.next();
                }

                return result;
        }
}
```

Example: A-7 Code for BulletinSubscriber.java

```
package com.ibm.itso.grid.gt3.bulletin.client;

import com.ibm.itso.grid.gt3.bulletin.common.stubs.service.BulletinServiceGridLocator;
import com.ibm.itso.grid.gt3.bulletin.common.stubs.BulletinPortType;
import com.ibm.itso.grid.gt3.bulletin.common.stubs.MessageDataType;

import org.globus.ogsa.utils.AnyHelper;
import org.globus.ogsa.NotificationSinkCallback;
import org.globus.ogsa.impl.core.service.ServicePropertiesImpl;
import org.globus.ogsa.client.managers.NotificationSinkManager;

import org.gridforum.ogsi.HandleType;
import org.gridforum.ogsi.ExtensibilityType;
import org.gridforum.ogsi.ServiceDataValuesType;

import java.net.URL;
import java.rmi.RemoteException;
import java.util.List;
import java.util.ArrayList;

public class BulletinSubscriber extends ServicePropertiesImpl implements
NotificationSinkCallback {
```

```
        List instanceNames;
        BulletinPortType[] bulletins;

        public static final void main(String[] args) {
                BulletinSubscriber bs = new BulletinSubscriber();
                bs.run(args);
        }

        private void run(String[] args) {

                // The base GSH and the instance names will be informed
                // as command-line arguments
                String baseGSH = args[0];

                // Build service instance references
                int instanceCount = args.length-1;
                BulletinPortType[] bulletins = new BulletinPortType[instanceCount];
                instanceNames = new ArrayList(instanceCount);

                try {
                        NotificationSinkManager notifManager =
NotificationSinkManager.getManager();
                        String sinks[] = new String[instanceCount];

                        for (int i=0; i<instanceCount; i++) {
                                String instanceName = args[i+1];
                                String GSHstr = baseGSH+"/"+instanceName;
                                URL GSH = new URL(GSHstr);
                                BulletinServiceGridLocator bulletinGL = new
BulletinServiceGridLocator();
                                bulletins[i] = bulletinGL.getBulletinServicePort(GSH);


notifManager.startListening(NotificationSinkManager.MAIN_THREAD);
                                sinks[i] = notifManager.addListener("ApprovedMessages", null,
new HandleType(GSHstr), this);

                                instanceNames.add(instanceName);
                                System.out.println("Subscribed to the \""+instanceName+"\"
instance.");
                        };

                        System.out.println("Type any key to exit.");
                        System.in.read();

                        // Stop listening
                        for (int i=0; i<instanceCount; i++) {
                                notifManager.removeListener(sinks[i]);
```

```
                        }

                        notifManager.stopListening();
                        System.out.println("Not listening anymore!");
                } catch (Exception e) {
                        e.printStackTrace();
                }
        }

        /* (non-Javadoc)
         * @see
org.gridforum.ogsi.NotificationSink#deliverNotification(org.gridforum.ogsi.ExtensibilityType)
         */
        public void deliverNotification(ExtensibilityType any) throws RemoteException {
                ServiceDataValuesType serviceData = AnyHelper.getAsServiceDataValues(any);
                MessageDataType mdt = (MessageDataType)
AnyHelper.getAsSingleObject(serviceData, MessageDataType.class);

                System.out.println("New message received.\n"+mdt.getMessage());
        }
}
```

Example: A-8 Code for BulletinWriter.java

```
package com.ibm.itso.grid.gt3.bulletin.client;

import com.ibm.itso.grid.gt3.bulletin.common.stubs.service.BulletinServiceGridLocator;
import com.ibm.itso.grid.gt3.bulletin.common.stubs.BulletinPortType;

import java.io.BufferedReader;
import java.io.InputStreamReader;
import java.net.URL;
import java.util.List;
import java.util.ArrayList;

public class BulletinWriter {

        public static final void main(String[] args) {
                // The base GSH and the instance names will be informed
                // as command-line arguments
                String baseGSH = args[0];

                // Build service instance references
                int instanceCount = args.length-1;
                BulletinPortType[] bulletins = new BulletinPortType[instanceCount];
                List instanceNames = new ArrayList(instanceCount);
```

```
                try {
                        for (int i=0; i<instanceCount; i++) {
                                String GSHstr = baseGSH+"/"+args[1+i];
                                URL GSH = new URL(GSHstr);
                                BulletinServiceGridLocator bulletinGL = new
BulletinServiceGridLocator();
                                bulletins[i] = bulletinGL.getBulletinServicePort(GSH);
                                instanceNames.add(args[i+1]);
                        };

                        boolean go = true;
                        while (go) {
                                BufferedReader br
                                                = new BufferedReader(new
InputStreamReader(System.in));

                                System.out.println("Enter the type of message you want to
submit");

                                String instanceName = br.readLine();
                                int instanceIndex = instanceNames.indexOf(instanceName);

                                if (instanceIndex == -1) {
                                        System.out.println("There isn't any service instance
named "+instanceName);
                                } else {
                                        System.out.println("Enter the message");
                                        String message = br.readLine();


bulletins[instanceIndex].submitApprovedMessages(message);
                                }

                               System.out.println("Do you want to submit another message (y/n)
?");
                                char key = Character.toLowerCase((char) br.read());
                                if (key != 'y') {
                                        go = false;
                                }
                        }
                } catch (Exception e) {
                        e.printStackTrace();
                }
        }
}
```

Example: A-9 Code for BulletinAdminConsole.java

```
package com.ibm.itso.grid.gt3.bulletin.client;

import org.globus.ogsa.utils.GridServiceFactory;
import org.gridforum.ogsi.Factory;
import org.gridforum.ogsi.LocatorType;
import org.gridforum.ogsi.OGSIServiceGridLocator;

import java.net.URL;
import java.io.*;
import java.util.*;


public class BulletinAdminConsole {

        public static void main(String[] args) {
                String addr = new String();
                if(args.length != 1) {
                        System.out.println("Usage: java
com.ibm.itso.grid.gt3.bulletin.client.BulletinAdminConsole <GSH>");
                        return;
                } else {
                        addr = args[0];
                }

                System.out.println("WELCOME TO BULLETIN ADMIN CONSOLE...");
                System.out.println("Command Help...");
                System.out.println(">c <instance name>   : to create a instance");
                System.out.println(">d <instance name>   : to destroy a instance");
                System.out.println(">q                   : to exit this program");
                System.out.println();


                try{
                        //Get command-line argument
                        URL GSH = new java.net.URL(addr);

                        //Get a reference to the Bulletin Service Factory
                        OGSIServiceGridLocator ogsiServiceGridLocator = new
OGSIServiceGridLocator();
                        Factory factory = ogsiServiceGridLocator.getFactoryPort(GSH);
                        GridServiceFactory gridServiceFactory = new
GridServiceFactory(factory);

                        BufferedReader buffer = new BufferedReader(new
InputStreamReader(System.in));
                        Hashtable instanceStore = new Hashtable();
```

```
                        System.out.println("please input command...");

                        while(true) {
                                System.out.print(">");
                                String command = buffer.readLine();
                                String name = new String();
                                if(command.length() > 0) {
                                        if(command.substring(0, 1).compareTo("c") == 0 &&
command.substring(2).length() >0) {
                                                name = command.substring(2);
                                                //Get a new Bulletin Service instances
                                                LocatorType locatorType =
gridServiceFactory.createService(name);
                                                instanceStore.put(name, locatorType);
                                                System.out.println("create " + name);
                                        } else if(command.substring(0, 1).compareTo("d") == 0
&& command.substring(2).length() >0) {
                                                name = command.substring(2);
                                                //Destroy a Bulletin Serivce instances
                                                LocatorType locatorType = (LocatorType)
instanceStore.remove(name);
                                                if(locatorType != null) {

ogsiServiceGridLocator.getGridServicePort(locatorType).destroy();
                                                        System.out.println("Instance " + name +
"destroyed");
                                                } else {
                                                        System.out.println("There's no instance
called " + name);
                                                }
                                        } else if(command.compareTo("q") == 0) {
                                                System.out.print("All instances are going to be
killed before exit, continue? <y/n>");
                                                command = buffer.readLine();
                                                if(command.compareTo("y") == 0) {
                                                        System.out.println("kill all
instances");
                                                        Enumeration elements =
instanceStore.keys();
                                                        while(elements.hasMoreElements()) {
                                                                name = (String)
elements.nextElement();
                                                                System.out.println("Instance "
+ name + "destroyed");
                                                                LocatorType locatorType =
(LocatorType) instanceStore.get(name);

ogsiServiceGridLocator.getGridServicePort(locatorType).destroy();
                                                        }
```

```
                                                        System.out.println("Bye");
                                                        return;
                                                }
                                        }
                                }
                        }
                } catch (Exception e) {
                        System.err.println("There was error while create/destroy instance");
                        e.printStackTrace();
                }
        }
}
```

B

Web service development

This appendix introduces the reader to the steps involved in developing a Web service based application. Since a grid service is an extension of a Web service, we present the development of Web services in this chapter as a prelude to our discussion of the development of grid services, which can be found in the later parts of this document.

Introduction

The concept of Web services is essential for grid applications, since it acts as the main mechanism that provides the interoperability grid needs.

Thus, this appendix presents the main tasks involved in the process of developing a Web service. It is worthwhile to mention that many tools used in this process may also be used in the development of grid services. However, a grid has other requirements that modify the development process, making necessary the use of specific tools.

There are several tools available on the market to assist in the development of Web services. We will focus on the development using the Axis tool. We have chosen Axis for the following reasons. First, Axis is an open source tool and popular in the development community. Second, GT3 uses Tomcat and Axis as the default container. Third, grid service leverages many features from Axis. Hence, familiarity with Axis will immensely aid the reader in understanding the development of grid services using GT3 as presented in subsequent sections of this document.

We present the various steps in building a Web service using the Axis tool with a simple application. The same application is used in the next part of the document where we discuss the development of a simple grid service. The use of the same application will help in clearly illustrating the extensions needed for a Web service to make it a grid service.

Our main focus is to illustrate the steps needed in building a Web service. We first present the problem statement that deals with the requirements of the application. Then, we discuss the individual steps in the development, deployment, and monitoring of the Web service application using the Axis tool.

Development tools

In this section, we will briefly introduce the open source and openly available tools that can be used for the Web services development. We have used these tools for developing the application discussed in this chapter.

Tomcat

Tomcat is a servlet container. It is a reference implementation of JSP and servlet technologies, and most of the samples in this appendix are running on Tomcat. More information on Tomcat can be found at:

http://jakarta.apache.org/tomcat/index.html

Axis

Axis is an implementation of the SOAP specification. As a SOAP engine, it is compliant with SOAP1.1/1.2. At present, GT3 implementation takes advantage of the Axis features. Axis is used throughout the document to develop the samples and will be elaborated upon in later sections. More information on Axis can be found in Appendix C, “Java2WSDL and WSDL2Java” on page 231 and at:

http://ws.apache.org/axis/

ant

ant is an open source, Java based build tool which helps in building the development project automatically. Besides the standard tasks shipped with **ant**, Axis and GT3 have their own **ant** tasks. Using these tasks facilitates development work. More information on **ant** can be found in Appendix D, “Tasks using ant” on page 237 and at:

http://ant.apache.org

Web services development basic steps illustrated

This section presents the tasks involved in Web services development, and illustrates them with a sample distributed client-server application.

The main function of the server is to return ‘the message of the day’ to any client that invokes it. Subsequent invocation of the server by the clients returns a different message. The mechanism of choosing the next message is to be determined by the server.

The major steps concerning a Web service development are as folllows:

Specifying	Phase to define the functionalities.
Coding	Phase to generate and adapt the code, WSDL and Java, using tools like Java2WSDL.
Building	Phase to compile the code WSDL, using WSDL2Java and Javac.
Deploying	Phase to deploy the WSDD files, using Java.
Testing	Phase to test the application; this can be done through some test code or some tool, like tcpmon.

The next sections explain each one of these phases.

Specifying

This simple application will be developed into two parts, namely a client part and a server part. The functionality of the application will be provided by a single Web service. A single operation of the Web service will provide the message of the day. The server will be the service provider and implement the aforementioned functionality. The client will be the service requestor and request the service from the service provider by binding to the service and invoking the operation on the Web service.

Coding

The specification of the Web services within the application must generate the code. There are two ways in which the Web service(s) can be coded:

1. The service can be specified as a Java interface. Axis provides a tool to generate the WSDL and the stubs that are necessary for the application execution from the Java interface.
2. The service can be specified within a WSDL file. The WSDL files can be created by the developer manually or obtained from a service registry, such as the UDDI. In this case, the code is ready for the Building phase.

Generating WSDL from a Java interface

In this section, we will illustrate the steps for creating a WSDL file from a given Java interface.

Defining the Java interface

Defining the Java interface is the first step in this process. Figure B-1 shows the Java interface for our example application defined in the previous section.

```
package com.ibm.itso.web.axis.sample;
public interface MOTD {
    public java.lang.String getMOTD();
}
```

Figure B-1 Interface: MOTD.java

After definition of the interface, the file must be compiled. It should be noted that an interface which inherits from another interface can also be defined for the service. The Axis tool allows an inheriting interface to be used as an input for the Java2WSDL tool.As specified before, the tool allows selection methods from the inheritance chain to be included in the generated Web service definition.

Generating the WSDL file

In this step, the WSDL file is generated from the Java interface definition. The following is the command line invocation of the tool for the generation of the WSDL file for our example application:

```
java org.apache.axis.wsdl.Java2WSDL -o itso.wsdl -l
"http://localhost:8080/axis/services/MOTD"
-n"http://sample.axis.web.itso/MOTD"
-p"com.ibm.itso.web.axis.sample"="http://sample.axis.web.itso/MOTD"
com.ibm.itso.web.axis.sample.MOTD
```

The use of the following options in the above invocation must be noted:

1. The output WSDL file name is specified as itso.wsdl using the `-o` option. The output file is stored under the current directory.
2. The service URL is specified as `http://localhost:8080/axis/services/MOTD` using the `-l` option.
3. The target namespace is specified as `http://sample.axis.web.itso/MOTD` using the `-n` option.
4. The package to namespace mapping pair is specified as `com.ibm.itso.web.axis.sample http://sample.axis.web.itso/MOTD` using the `-p` option.

Analysis of the generated WSDL file

In this section, we will discuss the various segments of the generated WSDL based in the specification WSDL 1.1 generated by the Axis tool. The WSDL 1.2 specification, still in draft at the time of the prodution of this document, has proposed extensions that include additional definition of grid services.

Figure B-2 on page 218 shows the WSDL file that is generated as a result of the previous step:

```
<?xml version="1.0" encoding="UTF-8"?>
<wsdl:definitions
     targetNamespace="http://sample.axis.web.itso/MOTD"
     xmlns:impl="http://sample.axis.web.itso/MOTD"
     xmlns:intf="http://sample.axis.web.itso/MOTD"
     xmlns:apachesoap="http://xml.apache.org/xml-soap"
     xmlns:wsdlsoap="http://schemas.xmlsoap.org/wsdl/soap/"
     xmlns:soapenc="http://schemas.xmlsoap.org/soap/encoding/"
     xmlns:xsd="http://www.w3.org/2001/XMLSchema"
     xmlns:wsdl="http://schemas.xmlsoap.org/wsdl/"
     xmlns="http://schemas.xmlsoap.org/wsdl/"
     >

  <wsdl:message name="getMOTDRequest">
  </wsdl:message>

  <wsdl:message name="getMOTDResponse">
    <wsdl:part name="getMOTDReturn" type="xsd:string"/>
  </wsdl:message>

..........
</wsdl:definitions>
```

Figure B-2 itso.wsdl file

In the previous section, we had specified for one WSDL file to be generated. Hence, the above WSDL file contains both the interface and the implementation specification. We will explain more about separating the interface and implementation into two WSDL files in the next section.

Figure B-3 below shows the port type definition in the generated WSDL.

```
<wsdl:portType name="MOTD">
  <wsdl:operation name="getMOTD">
    <wsdl:input name="getMOTDRequest" message="impl:getMOTDRequest"/>
    <wsdl:output name="getMOTDResponse" message="impl:getMOTDResponse"/>
  </wsdl:operation>
</wsdl:portType>
```

Figure B-3 PortType in itso.wsdl

The name of this port type is the same as the name of the interface. The name can be changed by using the **-P** option. The port type contains only one operation:getMOTD. It is the same as the function name defined in interface. The input and output messages, namely impl:getMOTDRequest and impl:getMOTDResponse respectively, for the operation are specified in the

figure. The descriptions of the messages, including the data types of the parts of the message, are specified in Figure B-3 on page 218. The messages and data types are specified before the description of the port type.

Figure B-4 below shows the implementation specification for the Web service. The implementation describe the bindings for a particular transport. In our example, the bindings are specified for the SOAP protocol.

```
<wsdl:binding name="MOTDSoapBinding" type="impl:MOTD">
  <wsdlsoap:binding style="rpc"
       transport="http://schemas.xmlsoap.org/soap/http"/>
  <wsdl:operation name="getMOTD">
    <wsdlsoap:operation soapAction=""/>

    <wsdl:input name="getMOTDRequest">
      <wsdlsoap:body use="encoded"
           encodingStyle="http://schemas.xmlsoap.org/soap/encoding/"
           namespace="urn:itso"/>
    </wsdl:input>

    <wsdl:output name="getMOTDResponse">
      <wsdlsoap:body use="encoded"
           encodingStyle="http://schemas.xmlsoap.org/soap/encoding/"
           namespace="urn:itso"/>
    </wsdl:output>

  </wsdl:operation>
</wsdl:binding>
```

Figure B-4 Binding part of itso.wsdl

As can be seen in the figure, the name of the binding is MOTDSoapBinding. The tool uses the default naming convention of the service port name followed by SOAPBinding in generating the binding name. The name can be changed by using the `-b` option. The default binding style used by Axis and also in the above definition is RPC. The name can be changed by using the `-y` option.

The Service Implementation definition for our example service is shown in Figure B-5 on page 220.

```
<wsdl:service name="MOTDService">
  <wsdl:port name="MOTD" binding="impl:MOTDSoapBinding">
    <wsdlsoap:address
         location="http://localhost:8080/axis/services/MOTD"/>
  </wsdl:port>
</wsdl:service>
```

Figure B-5 Service implementation definition in itso.wsdl

The default service name has the format of service port name followed by the word `Service`. The default name has been used by the Axis tool in generating the above service implementation specification. The service name can be changed by using the `-s` option. The service port name is specified in the location set variable and can be changed using the `-S` option. As can be seen from Figure B-5, our example service utilizes the SOAP bindings and can be accessed via the address `http://localhost:8080/axis/services/MOTD`.

Advanced options for WSDL generation

As was discussed earlier, two WSDL files can be generated for a single Java interface definition. When two files are generated, the first file contains the interface definition of the Web service while the other contains the implementation definition. The interface defines the contract of the Web service provided by the service provider. The implementation part defines the implementation specifics of the service. The same contract can be implemented in different ways depending on the implementation and protocol chosen. The interface WSDL file can be generated by specified by using `-w` and setting it to the value `Interface`. Similarly, the implementation WSDL file can be generated by setting the value of the `-w` option to `implementation`. The interface WSDL contains the type, message and port type definitions of the Web service. The implementation WSDL contains the bindings and services definition.

The Java interface that is used to generate the WSDL file can potentially inherit from other interfaces. In fact, it could be the leaf of an inheritance tree of Java interfaces. When an interface inherits from other interfaces, the Axis tool navigates the inheritance chain and includes all the inherited methods in the WSDL definition. The inherited methods are included in addition to the methods directly defined in the given Java interface. However, the tool can be instructed to ignore the inherited methods by using the `-c` option.

Building

This phase generates the client stub and server skeleton. As will be shown later in this chapter, Axis provides tools to generate the client side stub and the server

side skeleton from the service specification. The stubs provide the necessary runtime for the client program to communicate with the server in the programming language of the client. The server skeleton allows the server program to be transparent of the client programming language. It captures the metadata, assists the runtime in data conversions, and forwards the incoming service requests to the server implementation.

Moreover, the server side and the client side functions of the Web service are developed in Java, compiled and packaged. Axis provides the necessary tools.

Generating Java code from a WSDL file

In order to develop a Web service, its interface must be defined in a WSDL file. Subsequently, the client and server side code are implemented based on the WSDL definition. In some instances, an existing server side implementation is used. When an existing server implementation is used or is wrappered to develop a Web service, a previously defined Java interface can be used in the generation of the WSDL definition file. In the previous section, we discussed the process of creating a WSDL definition from a Java interface definition.

Besides generating a WSDL from a Java interface, the service provider can choose to create it manually or with some other tool, such as the WebSphere Studio Application Developer (WSAD). Since WSDL is much more expressive than a Java interface definition, some operations or requirements that cannot be expressed in Java can be expressed in WSDL. Hence, it is necessary in some instances to define the WSDL file directly.

Developers implementing the service requestor interface can primarily obtain the WSDL file in the following ways:

1. From a UDDI registry. The service requester can search the UDDI to find the appropriate service provider and the corresponding WSDL.
2. From service providers directly. For example, a URL, such as `http://localhost:8080/axis/services/MyService?WSDL` for a WSDL can be obtained from the service provider. Subsequently, the WSDL file can be accessed via a browser and saved.

In this section, we will focus on the process of creating the client-side stub and server-side skeleton Java code from a given WSDL definition. We will illustrate the process using the example WSDL that was generated in the previous section.

Generating stub and skeleton code from WSDL

`WSDL2Java` is a utility within the Axis toolkit that uses the information specified in a given WSDL file and generates Web service skeleton and stub code.

The client stub acts as a proxy and enables the client program to invoke grid services in their own programming language. Subsequently, the client stub transforms the data and the message provided by the client into the data format and protocol specified in the WSDL for communicating with the server.

The server skeleton captures the metadata of the service based on the WSDL information and provides it to the runtime for converting the incoming data and message into the format suitable for server implementation. Further, it forwards the incoming request to the server implementation.

The following is the command line invocation of the **WSDL2Java** utility for generating the stub and the skeleton for the example WSDL that was generated in the previous section.

```
java org.apache.axis.wsdl.WSDL2Java -o . -d Session -s -S true
-N"http://sample.axis.web.itso/MOTD"="com.ibm.itso.web.axis.sample"
itso.wsdl
```

The use of the following options in the above invocation must be noted:

1. The location of the generated code is specified using the `-o` option. As specified above, the code will be generated in the current directory.
2. The scope of the service deployment is specified using the `-d` option. As specified above, the service will be deployed within a Session scope.
3. The `-s` option instructs the utility to generate server side skeleton code as well.
4. The `-S` option instructs the utility to deploy the server side skeleton within the Axis environment.
5. The Java package to the namespace mapping pair is specified using the `-p` option. In our example, the package com.ibm.itso.web.axis.sample is mapped to `http://sample.axis.web.itso/MOTD namespace`.

Analysis of the generated files

Figure B-6 below lists the various files that are generated as a result of the execution of the utility specified in the previous section.

```
$>ls
deploy.wsdd  MOTDService.java          MOTDSoapBindingSkeleton.java
MOTDServiceLocator.java   MOTDSoapBindingStub.java
MOTD.java    MOTDSoapBindingImpl.java  undeploy.wsdd
```

Figure B-6 Generated files from WSDL

The following list briefly describes the content of each of the generated files:

1. The MOTDService.java file contains the interface of our service and is implemented by the MotDServiceLocator class.
2. The MOTDServiceLocator.java file contains the implementation of the MOTDServiceLocator class and aids the client in locating the reference to the service before invoking any operations on the service.
3. The MOTDSoapBindingSkeleton.java file contains the server side skeleton code.
4. The MOTDSoapBindingStub.java file contains the client side stub code.
5. The MOTD.java file contains the Java interface for the service to be used by both the client and the server implementation.
6. The MOTDSoapBindingImpl.java file contains a scaffold for the server implementation. The developer can implement the server side implementation code in this file.
7. The deploy.wsdd file contains the XML based deployment descriptor that is used for deploying the service in the container.
8. The undeploy.wsdd file contains the XML based deployment descriptor that is used for undeploying the service from the container.

It should be noted that our example service does not contain any complex data types. If a service requires a complex data type then the schema of the data type must be specified in the WSDL file. If a complex data type is defined within the WSDL, the utility additionally generates a Java bean class for each of the data types specified.

Implementing the server side code

In this section, we will elaborate on the server side implementation of our example Web service. As was stated in the previous section, the MOTDSoapBindingImpl.java file generated by the **WSDL2Java** utility provided the scaffold for implementing the server functionality. Figure B-7 on page 224 shows the content of the MOTDSoapBindingImpl.java file with our added implementation. As can be seen in the figure, the MOTDSoapBindingImpl class implements the MOTD interface that was generated. We define a private String array variable msgs to store a set of predefined messages. The implementation of the getMOTD() method retrieves the next message from the array and returns it to the client.

Note: The code in the Web service below uses a static int variable, making the service stateful. This is generally discouraged because the Web service container is allowed to create as many instances of the service as it desires, and connect any incoming request with any instance. This would cause chaos for most stateful requester-provider relationships, but in the case of our trivial example, there is no such thing as an “incorrect” message from the list being delivered to a requester.

```
package com.ibm.itso.web.axis.sample;
public class MOTDSoapBindingImpl implements
com.ibm.itso.web.axis.sample.MOTD{
   private static int lastOne=0;
   private static String msgs[]={
   "And 1.1.81 is officially BugFree(tm), so if you receive any bug-reports
   on it, you know they are just evil lies.",
   "As usual, this being a 1.3.x release, I haven't even compiled this
   kernel yet.  So if it works, you should be doubly impressed.",
   "How should I know if it works?  That's what beta testers are for.  I
   only coded it.",
   "I've run DOOM more in the last few days than I have the last few months.
   I just love debugging ;-)",
   "If you want to travel around the world and be invited to speak at a lot
   of different places, just write a Unix operating system.",
   "We all know Linux is great...it does infinite loops in 5 seconds.",
   "When you say 'I wrote a program that crashed Windows', people just stare
   at you blankly and say 'Hey, I got those with the system, *for free*'",
   "...you might as well skip the Xmas celebration completely, and instead
   sit in front of your linux computer playing with the all-new-and-improved
   linux kernel version.",
   "I'm an idiot.. At least this one [bug] took about 5 minutes to find..",
   "World domination.  Fast"
};
    public java.lang.String getMOTD() throws java.rmi.RemoteException {
System.out.println("MOTDSoapBindingImpl ().getMOTD(): entry" );
        lastOne = lastOne % 10;
        String motd = msgs[lastOne];
System.out.println("MOTDSoapBindingImpl ().getMOTD(): Selected motd '" +
motd + "'" );
        lastOne++;
        return motd;
    }
}
```

Figure B-7 New MOTDSoapBindImpl.java

After writing the server side functionality, the MOTDSoapBindingSkeleton.java file is compiled with the aid of the Java compiler and necessary dependent files are included.

Implementing the client side code

In this section, we will present and discuss the client side implementation. The client is the service requestor and invokes the Web service provided by the server. Figure B-8 shows the client implementation for our example application that uses the client side stub generated by the **WSDL2Java** utility. As shown in the figure, the client locates the reference to the MOTD Web service by using the instance of the MOTDServiceLocator class. The instance reference to the MOTD service is obtained by invoking the getMOTD() method. The operation on the service is invoked on the binding reference of the service. The call to the getMOTD() method invokes the operation.

```
package com.ibm.itso.web.axis.sample;
import com.ibm.itso.web.axis.sample.MOTD
public class MOTDClient {
   public static void main(String [] args) {
   try{
      MOTD binding;
      binding = new MOTDServiceLocator().getMOTD();
      java.lang.String value = null;
      value =((MOTDSoapBindingStub)binding).getMOTD();
      System.out.println("The result is:"+value);
   }catch(Exception e ){
      e.printStackTrace();
   }
   }
}
```

Figure B-8 Sample code using stub code

Figure B-9 on page 226 shows the implementation of the client of our example application that does not use the generated client side stub. Instead, the client uses the JAVAX-RPC API.

```
package com.ibm.itso.web.axis.sample;
import org.apache.axis.client.Call;
import org.apache.axis.client.Service;
import javax.xml.namespace.QName;
public class SimpleClient{
   public static void main(String [] args) {
   try {
      String endpoint = "http://localhost:8080/axis/services/MOTD";
      Service service = new Service();
      Call call = (Call) service.createCall();
      call.setTargetEndpointAddress( new java.net.URL(endpoint) );
      call.setOperationName(new QName("http://sample.axis.web.itso/MOTD",
      "getMOTD"));
      String ret = (String) call.invoke( new Object[] { } );
      System.out.println("The result is:'" + ret + "'");
   } catch (Exception e) {
      System.err.println(e.toString());
   }
   }
}
```

Figure B-9 Sample code using JAVAX-RPC

As can be seen in the code of Figure B-9, the client generates the service and a generic call. Subsequently, it sets the endpoint address of the service and the particulars of the operation before invoking the service. As is evident from a comparison of the above two client implementations, the use of the generated client side stubs simplifies the programming. The client side stub hides the details of the composition of the operation from the client.

Deploying and testing the Web service

During this phase, the code developed along with the generated code and other necessary libraries is packaged and deployed in the container and server for execution. The deployment is tested and the client and server communication is monitored to ensure proper deployment and execution. Axis provides the necessary tools for this step as well.

Analyzing the generated WSDD files

As was shown in the prior section, the **WSDL2Java** tool generates two files, namely deploy.wsdd and undeploy.wsd, to assist in the deployment and un-deployment of the Web service, respectively. These files describe the various chosen options for deploying and un-deploying the Web service. The AdminClient utility provided by Axis manages the Web services and uses the descriptors files for deployment

and un-deployment of the Web service. Figure B-10 below shows the deployment descriptor generated for deploying our example.

```
<deployment xmlns="http://xml.apache.org/axis/wsdd/"
    xmlns:java="http://xml.apache.org/axis/wsdd/providers/java">
<!-- Services from MOTDService WSDL service -->
<service name="MOTD" provider="java:RPC" style="rpc" use="encoded">
      <parameter name="wsdlTargetNamespace"
value="http://sample.axis.web.itso/MOTD"/>
      <parameter name="wsdlServiceElement" value="MOTDService"/>
      <parameter name="wsdlServicePort" value="MOTD"/>
      <parameter name="className"
value="com.ibm.itso.web.axis.sample.MOTDSoapBindingSkeleton"/>
      <parameter name="wsdlPortType" value="MOTD"/>
      <parameter name="allowedMethods" value="*"/>
      <parameter name="scope" value="Session"/>
</service>
</deployment>
```

Figure B-10 Deploy.wsdd

As shown in the figure, the namespaces that define the various tags and values used in the descriptor are first defined. The definition of the service element is similar to the one in the WSDL file. Subsequently, the name of the service, the provider of the service, the style of communication between the client and the service and usage of the encoding are specified. Additional particulars are provided as parameter values.The target namespace for the service, service element, port type for the service, the name of the class to be invoked for dispatching the service requests, the port type of the Web service, the set of allowed methods to be invoked, and the scope of the service are specified.

Deploying the Web service

Figure B-11 shows the command to deploy our example Web service. The name of the deployment descriptor is provided as a parameter to the command.

```
$>java org.apache.axis.client.AdminClient deploy.wsdd
Processing file deploy.wsdd
<Admin>Done processing</Admin>
```

Figure B-11 Deploy Web service on Axis by AdminClient

It should be noted that the above command processes the deployment descriptor and registers the service within the Axis environment. Various implementation class files must be copied into the appropriate directory before the service is

made operational. In our example, the class files are copied to the Axis Webapp/WEB-INF/classes directory.

Testing the Web service application

Once the service is successfully deployed, the deployment can be verified and the service tested. The list of installed services can be browsed with the aid of a browser. Figure B-12 shows the list of services installed in our example installation.

Figure B-12 List installed Web services

After verifying that the necessary services are installed, the service can be tested. Figure B-13 shows the result of the execution of the client application programs discussed earlier.

```
$>java com.ibm.itso.ws.axis.sample.MOTDClient
The result is:We all know Linux is great...it does infinite loops in 5
seconds.
$>java com.ibm.itso.ws.axis.sample.SimpleClient
The result is:I'm an idiot.. At least this one [bug] took about 5 minutes to
find..
```

Figure B-13 The result of client executions

As can be seen, the execution of the client obtains the message of the day from the Web service and displays it on the screen as expected. In the next section,

we will demonstrate the use of the **tcpmon** utility for inspecting the messages exchanged between the server and the client.

Using tcpmon

In this section, we describe the functioning of a monitoring utility from Axis. The messages that are transferred between the service requestor and the service provider can be intercepted and inspected with the aid of the **tcpmon** utility. The tool can be launched by invoking the following command:

```
java org.apache.axis.utils.tcpmon
```

In order to view the SOAP messages that are transferred, the **tcpmon** must be set to listen on port 8081. The messages must be transferred from localhost:8080 to localhost:8081.The endpoint of the service must be modified to be on port 8081. Appropriate modifications to the service endpoint either in the deployment description or the client code must be made. The messages monitored by the **tcpmon** are displayed on the screen. Figure B-14 on page 230 shows the result of a monitoring session.

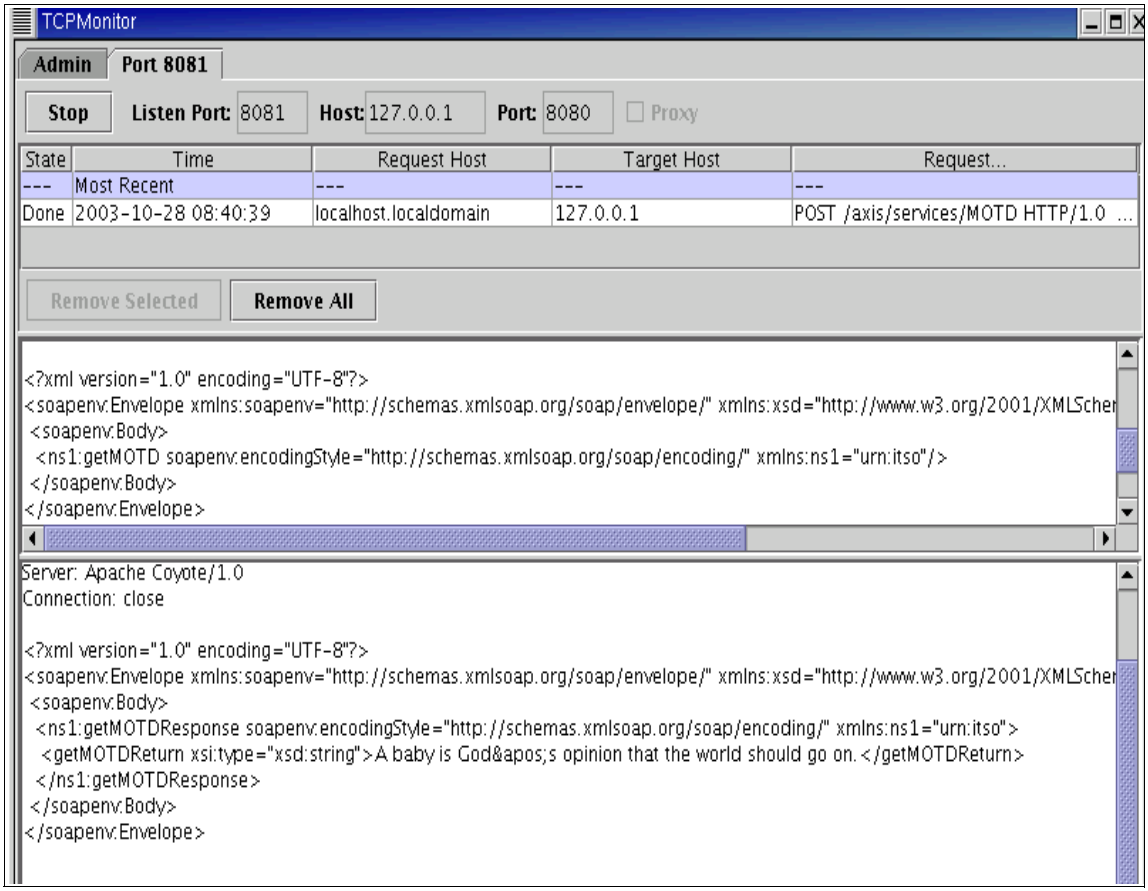


Figure B-14 SOAP messages between server and client

C

Java2WSDL and WSDL2Java

This appendix describes the key utilities that are part of the Axis toolkit, Java2WSDL and WSDL2Java. The use of these tools in building our Web services application is illustrated. In addition to the functionality of the individual utility, we will present the various options that can be specified in the invocation of the utility. Understanding of the options will aid the user in appreciating the finer details of the utility's function.

Java2WSDL

Java2WSDL is one utility in the Axis toolkit suite. It converts a Java interface into a WSDL file. In Java, a class to be implemented is specified using a Java interface. Since WSDL is more involved in its specification than a Java interface, this tool provides Java developers with a mechanism to generate Web service specifications in WSDL from a given Java interface.

This tool comes in handy when wrappering an existing Java implementation as a Web service. The Java interface defining all the public functions of the given class will be used as the input in generating the corresponding WSDL file for the service. The tool allows the user to selectively choose the methods to be exported from the Java interface. Hence, a user can only make a subset of the publicly available methods as Web service operations.

Java2WSDL can generate one or two WSDL file(s) based on user's input. If one output file is desired, then the generated WSDL file contains the port type, binding and service implementation specifications. On the contrary, when two output files are desired, the tools generates two WSDL files; the first file defines the interface and the second file defines the implementation.

The options to be used in the invocation of the utility and the corresponding explanations are shown in Table C-1.

Table C-1 Java2WSDL options

Option	Meaning
-l, --input <argument>	Define the input WSDL file
-o, --output <argument>	Define the output WSDL file
-l, --location <argument>	Define the location of the class file
-P, --portTypeName <argument>	Define the port type name. If is not specified, the class name is used by the utility instead
-b, --bindingName <argument>	Define the binding name. If it is not specified, the portName+"SOAPBingding" are used by the utility instead
-S, --serviceElementName <argument>	Define the service name. If not specified, using the portName+"Service"
-n, --namespace <argument>	Define the target NameSpace in target WSDL

Option	Meaning
-p, --PkgtoNS <argument>=<value>	Define the convert pair between package and NameSpace value
-m, --methods <argument>	Define the method(s) to export. Methods are separated by space or comma
-a, --all	Export all allowed methods in inherited class
-w, --outputWsdlMode <argument>	Define the output WSDL Mode: ALL, Interface or Implementation
-L, --locationImport <argument>	Define the place where the interface WSDL file located
-N, --namespaceImpl <argument>	Define the target namespace for implementation WSDL
-O, --outputImpl <argument>	Define the output implementation WSDL
-i, --implClass <argument>	Define the class that contain implementation code
-x, --exclude <argument>	Define the methods which will not be export. Commands are separated by space or comma
-c, --stopClasses <argument>	if --all is setting, -c define the class names that will stop inheritance search. names are separated by space or comma
-T, --typeMappingVersion <argument>	Define the value of sopaAction field. One of DEFAULT , OPERATION or NONE.
-y, --style <argument>	Define the WSDL binding style. One of DOCUMENT, RPC or WRAPPED
-u, --use <argument>	Define the binding style. One of LITERAL or ENCODED
-e, --extraClasses <argument>	Define the class names to be added, separated by space or comma
-C, --importSchema	Define an XML schema that will be imported into target WSDL

WSDL2Java

WSDL2Java is another utility in the Axis toolkit suite. As the name suggests, it converts a WSDL file into a Java file. Given a WSDL file with specifications of a Web service, it generates the necessary client stub. The client application invokes the client stub in order to communicate with the Web service. Additionally, with the use of appropriate options, it can generate server side skeleton code as well. The server skeleton dispatches the incoming service requests from the client to the server implementation.

The various options along with the corresponding explanations are shown in Table C-2.

Table C-2 WSDL2Java options

Option	Meaning
-n, --noImports	Do not generate code for the import WSDL
-O, --timeout <argument>	Define the time-out in second. (Default value is 45 while -1 mean disable)
-D, --Debug	Print debug information
-W, --noWrapped	Define support “wrapped” document/literal no not
-s, --server-side	Define generate server-side code
-N, --NStoPkg <argument>=<value>	Define the pair between namespace and package
-f, --fileNStoPkg <argument>	Define the property file that contain NameSpace and Package pairs
-p, --package <argument>	Define the package name used in generated code, omit all mappings
-o, --output <argument>	Define the output location of the generated code
-d, --deployScope <argument>	Define the scope of the deployment in deploy.wsdd. One of “Application”, “Request”, “Session”
-t, --testCase	Generate Junit testcase class
-a, --all	Generate all elements in WSDL file

Option	Meaning
-T, --typeMappingVersion <argument>	1.1 or 1.2. 1.1 mean SOAP 1.1 JAX-RPC. 1.2 mean SOAP 1.1 encoded
-F, --factory <argument>	Define the classname of the factory
-H, --helperGen	Generate helper classes
-U, --user <argument>	Define the user name for accessing the WSDL
-P, --password <argument>	Define the password for accessing the WSDL

D

Tasks using ant

This appendix introduces some of the key tasks provided by the **ant** - an Axis tool that helps to automate the build process. In a typical Web services development scenario, the developer has to periodically execute the various utilities. The manual process of entering the same commands several times can be laborious and error-prone. The **ant** tool is a Java based build tool similar to the **make** tool in Unix systems. Several targets, dependencies among the targets, and commands for the execution of tasks for building the target can be specified in a build file. The tool controls the build process based on the information specified in the build file. More information about **ant** can be found at:

http://ant.apache.org

axis-wsdl2java

The **axis-wsdl2java** task is similar in function to the **WSDL2Java** utility described earlier. When triggered, it generated the client stubs and server side skeletons from a WSDL description.

The various options to be used with the task are similar to the options of Axis utility **WSDL2Java** and are described in the table below.

Table D-1 axis-wsdl2java option

Options	Meaning	Default Value
all	Whether to generate a reference element or not	false
debug	Whether to generate debug output or not	false
deployscope	Define the scope in deploy.wsdd	
factory	Define a class that had implement GeneratorFactory interface	
helpergen	Define generate separate Helper classes or not	
namespacemappingfile	Define the map file which contain the map between package and namespace	NStoPkg.properties
noimports	Define generate code from imported WSDL files or not	false
output	Define output directory	
serverside	Define generate server side binding or not	false
skeletondeploy	Define use skeleton (true) or implementation (false) in deploy description file.	false
testcase	Define generate Junit testcase or not	false
timeout	Define the time-out for URL retrieval	45 second

Options	Meaning	Default Value
typemappingversion	Define the type mapping registry(1.1/1.2)	1.1
url	Define the place WSDL file located	
verbose	Define generate verbose output or not	false

The following shows the usage of the **axis-wsdl2java** task in the build file for our example development.

```
- <target name="generateJava">
- <axis-wsdl2java output="${src.dir}" deployscope="Session"
serverside="true" skeletondeploy="true" url="${WSDLFileName}">
  <mapping namespace="${targetNS}" package="${Package}" />
  </axis-wsdl2java>
  </target>
```

As can be seen, the target name is specified as `generateJava`. In order to build the target, the **axis-wsdl2java** must be executed. The various options for the tool are specified as well.

axis-java2wsdl

The **axis-java2wsdl** task is similar in function to the **Java2WSDL** utility described earlier. When triggered, it generated a WSDL description from a Java interface.

The various options to be used with the task are similar to the options of Axis utility **Java2WSDL** and are described in Table D-2.

Table D-2 axis-java2wsdl options

Option	Meaning	Default Value
bindingname	Define the binding name	servicePortName + "SoapBinding"
classname	Define the classname	
exclude	Define the methods that will not be excluded from WSDL file	

Option	Meaning	Default Value
extraclasses	Define the class names that will be used	
implclass	Define the implement class to get some debug information	
input	Define other WSDL which will be import into the destination	
location	Define the URL of service location	
locationimport	Define the location of interface WSDL	
methods	Define the methods that will be exported to WSDL file	
namespace	Define the target namespace	
namespaceimpl	Define the namespace of implement WSDL	
output	Define the name of output WSDL	
outputimpl	Define the name of output implement WSDL file	
porttypename	Define the name of port type	class name
serviceelementname	Define the service name	
serviceportname	Define the service portname	
stopclasses	Define the classes that stop inheritance search	
style	Define the style of WSDL. One of the following value: RPC, DOCUMENT, WRAPPED	

Option	Meaning	Default Value
typemappingversion	Define the default type mapping registry. One of the following value: 1.1, 1.2	1.1
use	Define the user name to access WSDL file	
useinheritedmethods	Define export the methods got from inheritance search	false

The following shows the usage of the **axis-java2WSDL** task in the build file for our example development

```
- <target name="generateWSDL" depends="compileInterface">
- <axis-java2wsdl classname="${PackageName}.${InterfaceName}"
namespace="${targetNS}" output="${WSDLFileName}"
location="${ServiceLocation}">
  <mapping namespace="${targetNS}" package="${Package}" />
</axis-java2wsdl>
  </target>
```

As can be seen, the target name is specified as `generateWSDL`. This task depends on another task, namely, compileInterface. In order to build the target genereateWSDL , **axis-java2WSDL** must be executed. The various options for the tool are specified as well. It should be noted that in order to build a target, the tool ensures that the depending targets are built first. Hence, in the usage above, the compileInterface target will also be built.

axis-admin

axis-admin is another **ant** task. This task can be used for administering the server. The options to be used with this task are shown in Table D-3.

Table D-3 Axix-admin options

Options	Meaning	Default
debug	Define working in debug mode or not	
failonerror	Define halt or not if fails	false
fileprotocol	Define that simple file protocol will be used	

Options	Meaning	Default
hostname	Define the hostname	
newpassword	Define the new password. Only working while action=passwd	
password	Define the password	
port	Define the port number	
servletpath	Define the path of admin servlet	
transportchain	Define the transportchain	
url	Define the URL of AxisServlet	
username	Define the username	
xmlfile	Define the XML file that will be processed	

E

Delegation

This appendix presents some complementary considerations about delegation and inheritance in grid development environments.

Delegation and operational providers

In the basic grid service presented in Chapter 4, “Grid services development” on page 37, grid services extend from the standard GridServiceImpl class. This is illustrated in Figure E-1.

This GridServiceImpl class contains the base functionality of a grid service. A new grid service inherits this base functionality from GridServiceImpl and extends that class by adding additional functionality which is specific to the application or this particular grid service.

The idea is that the set of base functionality that all grid services must implement is contained in the base GridServiceImpl class so that developers of new services only need to focus on new application specific business logic and not on implementing the standard grid service functions. This also isolates applications and developers from the GridService implementation which allows the base GridService implementation to change without requiring code changes for classes, which extend the base GridServiceImpl class.

Notice in Figure E-1 that the implementation class only contains application specific methods. All methods associated with a grid service implementation are encapsulated in the GridServiceImpl class which the implementation class extends.

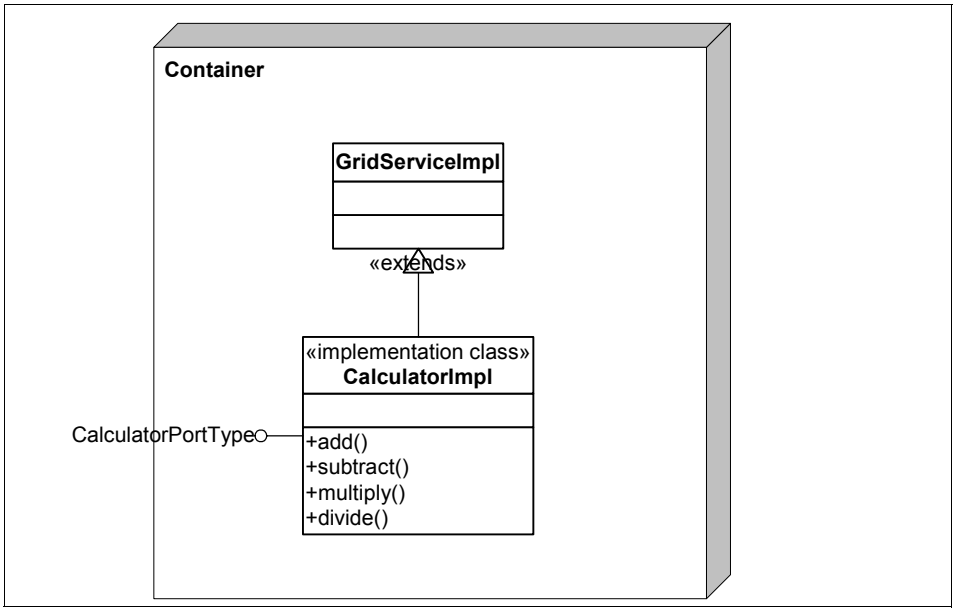


Figure E-1 Inheritance from GridServiceImpl

This is a standard Object Oriented Design (OOD) approach and was adequate for the basic grid service examples presented so far. However, it does have limitations.

Suppose that a developer wants to expose the functionality of an existing Java class as a grid service. This is a realistic scenario since there is an abundance of existing code and applications being integrated into the grid infrastructure. From the Java developer's perspective, the approach would be to extend the base Java class which contains the functionality the developer wishes to expose as a grid service. However, to implement a grid service, the previous section indicated that a grid service developer must extend the GridServiceImpl class to inherit the basic grid service functionality.

As seen in "Service implementation" on page 44, multiple inheritance is not allowed in Java or considered a good design approach; the grid service developer would need to find another solution to this problem. So how would a developer expose an existing Java class as a grid service?

The answer to this question is to use the delegation approach. The delegation approach solves this problem and provides a modular way for developers to distribute operations into several classes. Each of these classes is called an Operational Provider. Methods which provide similar operations can be grouped into functional categories and saved in an Operational Provider class which implements the Operational Provider interface.

In the following example, the calculator implementation (CalculatorImpl) methods have been grouped into basic calculator functions and scientific calculator functions. The functionally similar methods, in this case basic calculator methods or scientific calculator methods, are grouped into Operational Provider classes.

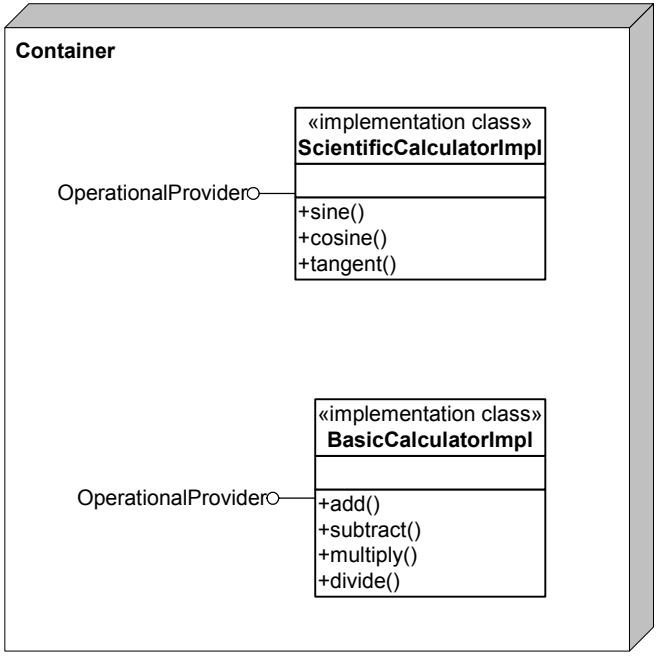


Figure E-2 Delegation using Operational Providers

Note that the classes in Figure E-2 do not extend any base class. They only have to implement the OperationalProvider interface. This would allow the grid service developer to extend an existing Java class to *gridify* it or encapsulate it in a grid service. The example shown in Figure E-3 on page 247 illustrates the inheritance approach and demonstrates that an inheritance implementation must extend GridServiceImpl, which precludes it from extending an existing base Java class, thereby turning it into a grid service.

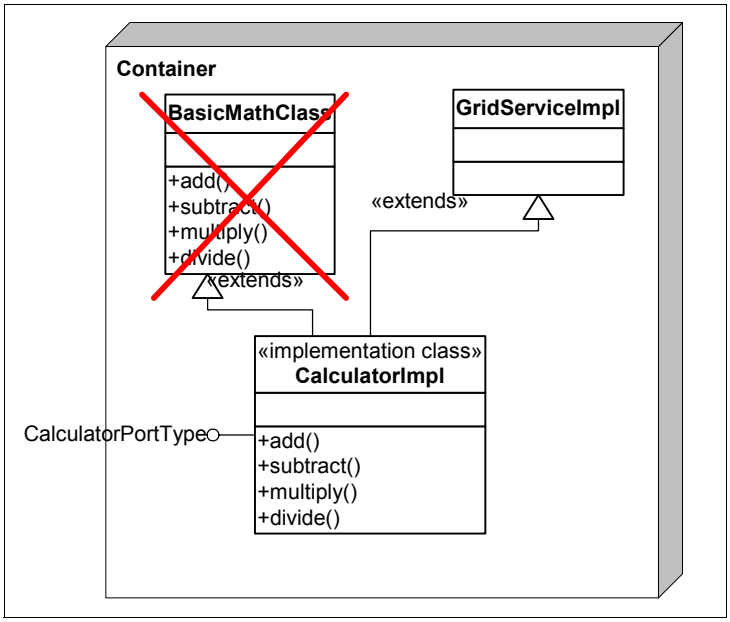


Figure E-3 Can not extend from an existing Java class and from GridServiceImpl

Figure E-4 on page 248 illustrates an Operational Provider class extending an existing Java BasicMath class. The CalculatorImpl can use an existing BasicMath class in its implementation, thus reducing the development and test time required. Note that in this approach, the implementation class does not have to extend the GridServiceImpl class.

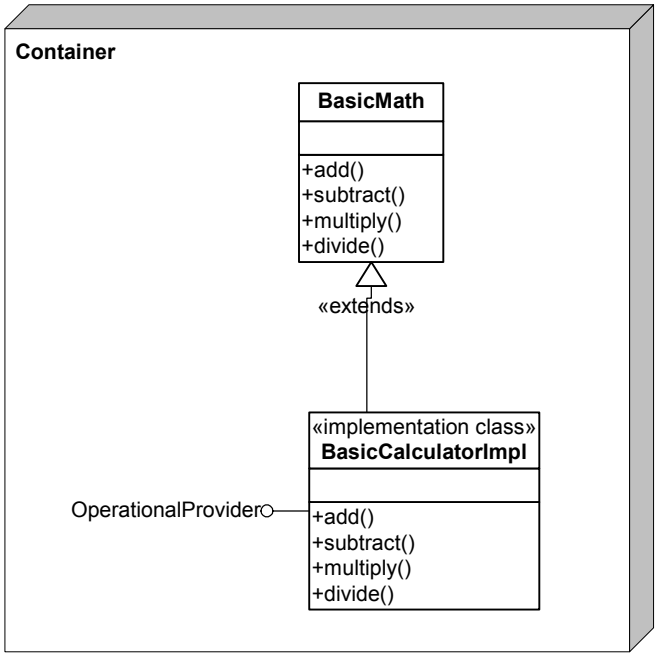


Figure E-4 The delegation model using Operational Providers supports the creation of a grid service from an existing Java class

The question comes to mind: what is providing the basic grid service functionality that was previously provided by GridServiceImpl? Does each Operational Provider class now need to implement the basic grid service functions that were previously provided by inheriting this functionality from the GridServiceImpl class? The answer is that GridServiceImpl functionality is still present and can be provided by the container. This frees the newly developed grid service from having to extend this GridServiceImpl class and allows the developer to focus on application specific business logic or extend an existing Java class to expose its functionality to the grid. Providing the GridServiceImpl functionality from the container is accomplished by updating the deployment descriptor to inform the container that it will provide the basic functionality of a grid service that was provided by GridServiceimpl in the inheritance approach.

The following line in the deployment descriptor handles this task:

Example: E-1 GridServiceImpl identified as the base class in the deployment descriptor

```
<parameter name="base-className"
value = org.globus.ogsa.impl.ogsi.GridServicImpl/>
```

As you see in Example E-1 on page 248, the value of the base-className parameter is `GridServiceIImpl`.

Now the grid service does not need to extend the GridServiceImpl class; it can simply implement the Operational Provider interface as show below in the Operational Provider Example E-2.

Example: E-2 Implementing Operational Providers

```
public class myclass implements OperationalProvider
```

By way of contrast to the inheritance method presented previously, see the following example.

Example: E-3 Inheritance from GridServiceImpl

```
public class myclass extends GridServiceImpl
```

As mentioned previously, the class must implement the Operational Provider Interface. The OperationalProvider interface contains two methods that need to be implemented in any implementation of that interface. These two methods rely on a private property called Operations. This private Operations property identifies what operations are provided by the class.

The two methods from the OperationalProvider interface that must be implemented are initialize and getOperations. The initialize method initializes the object while the getOperations method returns a list of operations that are provided by the class. The list of operations is returned in an array of QName[]. The list can be identified in several ways. If a class has a single operational provider then a wildcard can be used in its operations property, as shown in Example E-4:

Example: E-4 Single operational provider identified with a wildcard in the operations property

```
private static final QName[] operations = new QName[] {new QName("","*")
```

If the grid service has more than a single operation then each operation should be individually listed in the Operations property. See Example E-5 on page 250.

Example: E-5 Multiple operations, each listed individually

```
private static final QName[] operations = new QName[] {
new QName("", "method_name_1"),
new QName("", “method_name_2”),
new QName("", “method_name_3”),
…
 } ;
```

For example, the following Calculator class demonstrates an approach for listing each operation provided in the Operations property. Notice that the following operations are identified in Example E-6: add, subtract, multiply, and divide.

Example: E-6 Code sample showing multiple operations listed individually

```
package com.ibm.itso.gt3.providers.impl;

import org.globus.ogsa.GridServiceBase;
import org.globus.ogsa.GridServiceException;
import org.globus.ogsa.OperationProvider;
import org.globus.ogsa.GridContext;
import java.rmi.RemoteException;
import javax.xml.namespace.QName;

public class BasicCalculatorProvider implements OperationProvider {
   // Operation provider properties
   private static final QName[] operations =
                    new QName[]{new QName[] {
      ("", "add"),
      (““, “subtract”),
      (““, “multiply”),
      (““, “divide”))
};

   private GridServiceBase base;
}
   // Operation Provider methods
   public void initialize(GridServiceBase base) throws GridServiceException
   {
      this.base = base;
   }

   public QName[] getOperations()
   {
      return operations;
   }
```

```
    private int value = 0;

    public void add(int a) throws RemoteException
    {
        value = value + a;
    }

    public void subtract(int a) throws RemoteException
    {
        value = value - a;
    }

    public void multiply(int a) throws RemoteException
    {
        value = value * a;
    }

    public void divide(int a) throws RemoteException
    {
        value = value / a;
    }

    public int getValue() throws RemoteException
    {
        return value;
    }
}
```

The deployment descriptor would also need to inform the container that the implementation of the methods in the OperationalProvider interface will be found in the new grid service implementation of the OperationalProvider interface. This is accomplished by updating the deployment descriptor with the following lines as illustrated in Example E-7.

Example: E-7 Deployment descriptor update

```
<parameter name="instance-operationalProviders" value =
<path.Provider>/>
```

The deployment descriptor must also specify the port type of the new grid service. This is required since the new grid service does not explicitly declare that it implements the port type. This is accomplished by updating the deployment descriptor with the following line, shown in Example E-8 on page 252.

Example: E-8 Specifying the port type in the deployment descriptor

```
<parameter name="instance-className" value = <path.portType>/>
```

In summary, the following changes are required in the deployment descriptor (Example E-9).

Example: E-9 Changes required in the deployment descriptor

```
<parameter name="instance-className" value = <path.portType>/>

<parameter name="instance-operationalProviders" value = <path.Provider>/>

<parameter name="base-className"
value = org.globus.ogsa.impl.ogsi.GridServicImpl/>
```

Note that the changes to the grid service described above are all in the service implementation, which is transparent to the clients or users of this grid service.

F

Service Browser

This appendix presents a brief overview of the **`Service Browser`** utility. Working as a generic grid client application, this GUI tool might be used for testing and, to some extent, debugging the basic functionality of every grid service. Additionally, the fact that this tool is shipped as a standard GT3 component makes it a good choice for quick tests.

Introduction

The **Service Browser** tool has been conceived to work as a generic client-side application. Thus, it provides, by means of a user-friendly GUI, the basic operations that might be performed by any client in a grid service. Its most powerful feature, which is its ability to test remote method invocations from a runtime generated graphical form, makes it possible for services to be tested before any piece of real client code is written.

In addition to this feature, the **Service Browser** is capable of creating and destroying service instances, which might be useful for testing if a service has been correctly deployed, and there are facilities to check the contents of Service Data Elements.

In the next sections, all of these operations will be briefly introduced and, when applicable, demonstrated with a real grid service.

Basic operations

To start the **Service Browser**, simply type:

```
$ cd $GLOBUS_LOCATION
$ globus-service-browser
```

Figure F-1 Starting the Service Browser

Do this in the machine where the services to be tested have been deployed. Shortly, a graphical user interface similar to the one presented in Figure F-2 on page 256 should be displayed.

The main panel of the **Service Browser** has several sets of graphical controls. These sets can be briefly described as follows.

1. The set of buttons located at the top of the panel are the main controls of the **Service Browser**. Their functionality includes navigational facilities (Back and Forward buttons) and window management (New Window, Close and Refresh). The check box labeled `Show dynamic gui` is explained later in this section.
2. The text field located right beneath this first set of buttons is the field where you should specify the URI of the service to be inspected. When starting the **Service Browser**, the default service under inspection is shown. After setting a new service URI, the **Go** button has to be clicked for the panel to be updated.

3. Below the service URI text field there is a sub-panel with three tabs. Each tab, namely Services, WSDL and Service Data, presents a particular set of controls for monitoring and managing the service specified.
   a. The Services tab has four sets of controls:
      - The first one, enclosed in a frame named Message Security, is used to monitor and manage secure grid services as they address authorization and authentication issues
      - The second set contains controls that are used for querying the service data of a service as a way to search for services
      - The third set contains two tabs where the URIs of the services currently being monitored are displayed
      - The fourth, enabled only when the **Show dynamic gui** check box is selected, presents a set of fields and buttons that are dynamically generated according to the port type of the currently inspected service.
   b. The WSDL tab displays the WSDL definition of the service that is pointed to by the `instance-schemaPath` property of the service's deployment descriptor file.
   c. The Service Data tab contains a table where all the information about all the Service Data Elements of the service is presented. Additionally, a text area displays further information about the selected Service Data Element.

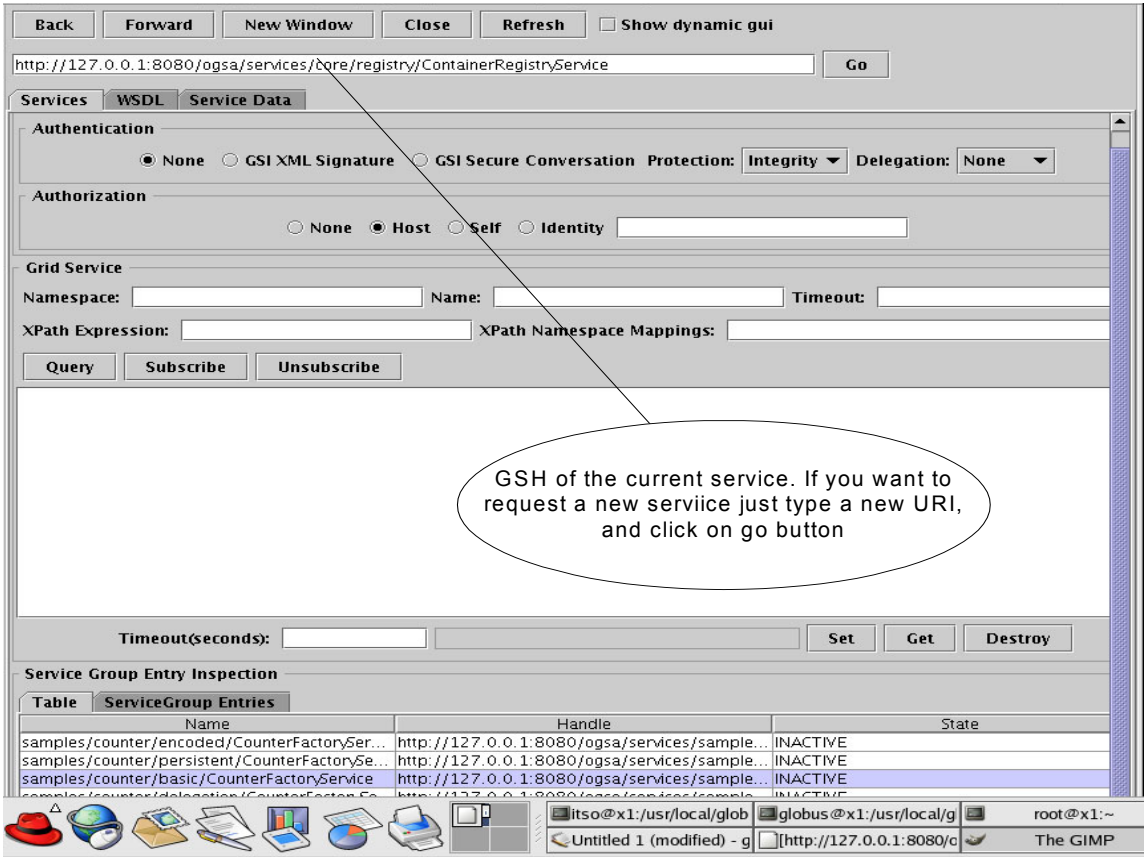


Figure F-2 The main Service Browser window

An alternative way to choose a particular service for inspection is to double-click its name in the table located under the Services tab. Doing so makes the panel update its state to the newly selected service and generates the proper dynamic GUI if the **Show dynamic gui** check-box is selected.

Figure F-3 on page 257 shows a screen shot taken from the portion of the window that houses the dynamically generated GUI. The entry that has been chosen to be inspected in this example was the **`MOTDFactory`**. As a factory, this service exports a method for the creation of service instances that can be invoked by clicking the button **Create Instance**.

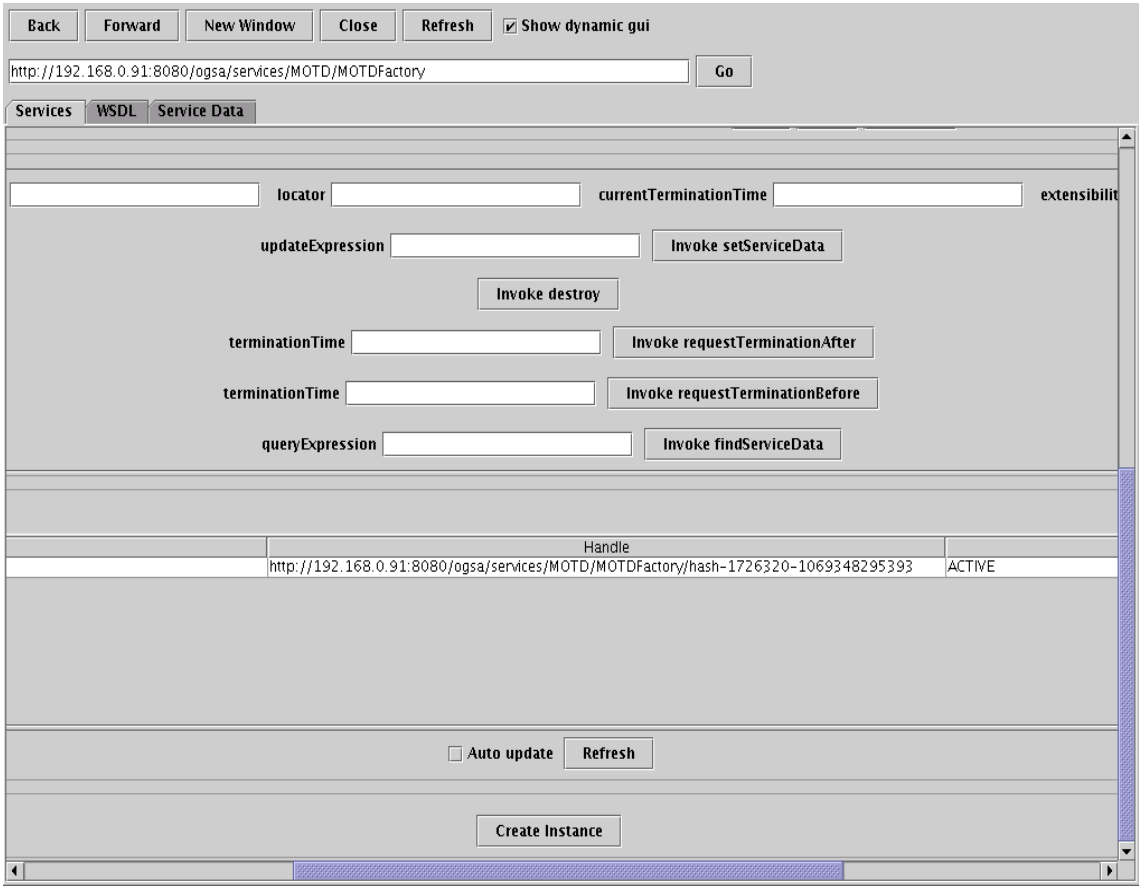


Figure F-3 Inspecting the MOTDFactory service with the Service Browser

When an instance is created, a new window similar to the main panel appears, displaying information about this instance (see Figure F-4 on page 258). Its name, which is built based on a random key appended to the name of its factory, is displayed in the top text field located right below the main controls. The remaining inspecting controls present further information about the service.

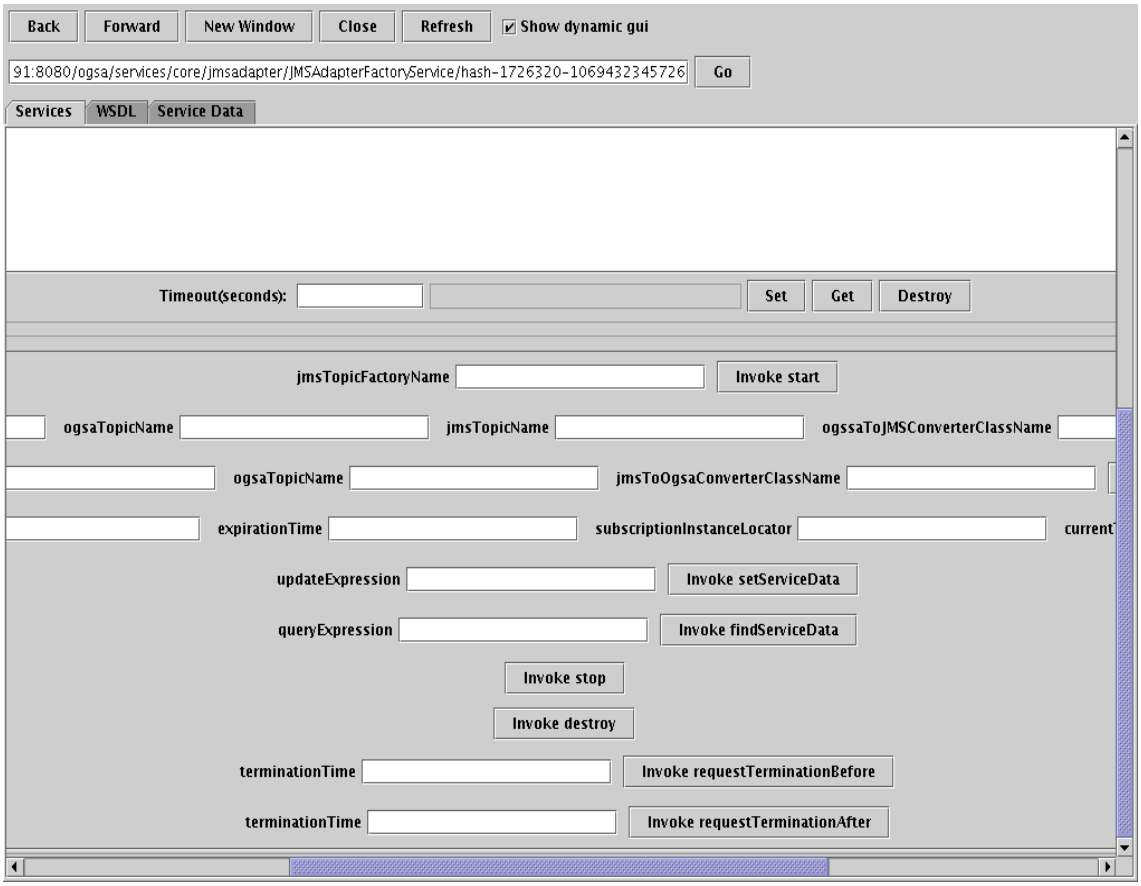


Figure F-4 Dynamic controls in the Service Browser

As you can see in Figure F-4, the multiple controls shown in the dynamically generated form refer to the standard operations exported by every port type. They might be used to set the termination time of the instance, to destroy the instance and to perform few other low-level operations, depending on how the port type was declared. Additionally, this form contains controls that refer to the specific operations declared in the service's port type. These controls might be used to test a particular implementation of the port type and check that the behavior of the service is correct.

As explained previously, if one of the two other tabs is selected, then you will be presented with information about the WSDL and the service data of this instance, respectively. In the case of the Service Data tab, it will be possible to visualize all the standard (common) Service Data Elements that are part of every port type.

The SDEs can be inspected for details about the data they store, namespace, etc.

The creation and destruction of service instances along with the inspection of general information about a service factory or a service instance are considered basic **Service Browser** operations. In the next section, we will discuss some of the advanced operations that might be performed with this tool.

Advanced operations

The advanced operations that can be performed by the **Service Browser** are the following.

Security monitoring and testing

When a service is selected for inspection, the contents of the frame labeled **Message Security** is updated with the security information of the service. These controls might be used to set the proper attributes needed by security enabled services during authorization and authentication.

For enabling security in a service, a set of specific configuration steps should be taken such as the issuing of certificates and the proper configuration of the service deployment descriptor file. Depending of the level of security required, the configuration steps might be lengthy and error-prone, making it difficult to test and debug the service package along with the client code. This section of the **Service Browser** allows for the test of these basic configuration requirements prior to the development of any client.

Service query

The section of controls right below the Message Security frame is the one through which services might be searched for by means of their service data. This search process is fairly similar to the one employed by the Index Service and can be described generally as a search on service data attributes.

For a search to be performed, you should enter the attributes to be queried in the provided text fields. If the namespace and name of a service are provided, then the **Service Browser** will look for an exact match for these names. An alternative way to perform a search is to provide an XPath expression; in this case, the search result will be all the services whose name and/or namespace match the specified pattern.

G

WSRF

This appendix provides an overview of the WS-Resource framework, known as WSRF, an open framework for modeling and accessing stateful resources using Web services. WSRF defines how Web service standards are evolving to meet grid services.

Introduction

Since Globus Toolkit V3.0 was released in July 2003, the GGF and the Globus Alliance have been working closely to define enhancements to the standards.

In January 2004, the WS-Resource Framework (WSRF) was presented, an open framework for modeling and accessing stateful resources using Web services. WSRF defines how Web service standards are evolving to meet grid services elements and requirements, as illustrated in Figure G-1. The specification is broken up into separate specifications, each focusing on a specific area. The document *From Open Grid Services Infrastructure to WS-Resource Framework: Refactoring & Evolution Version 1.0*, from 2/12/2004 (http://www.globus.org/) introduces the following normative WSRF specifications:

- WS-ResourceProperties: specifies stateful Web services
- WS-ResourceLifetime: specifies Web service life cycle
- WS-RenewableReferences: specifies Web service endpoint reference and addressing
- WS-ServiceGroup: specifies the creation and use of groups of Web services
- WS-BaseFault: specifies fault type used for fault error reporting
- WS-Notification: specifies the notification framework

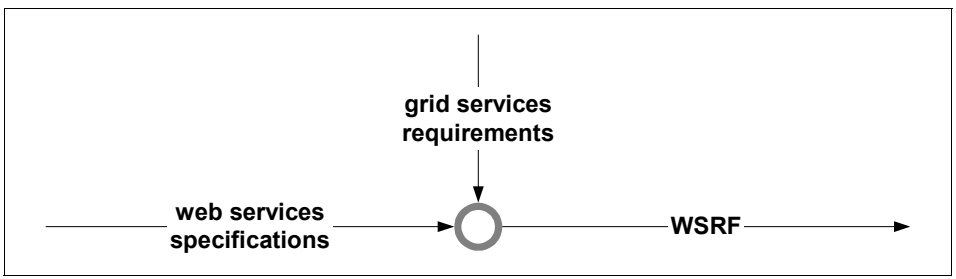


Figure G-1 Grid and Web service convergence

For more information about OGSA-WG (GGF Workgroup of OGSA) and WSRF, please refer to the following Web sites:

- http://www.globus.org/
- http://www.ggf.org/
- http://forge.gridforum.org/
- http://www.oasis-open.org/
- http://www.globusworld.org/
- http://www.ibm.com/developerworks/library/ws-resource/

WS-Resource Framework

The WS-Resource Framework is a set of six Web Services specifications that define terms such as the WS-Resource approach to modeling and managing state in a Web services context. Three drafts of specifications have been released at present, as well as an architecture document that describes the WS-Resource approach to modeling stateful resources with Web services. There are also plans for other related documents, one of them comparing the WS-Resource Framework with the Open Grid Services Infrastructure.

WSRF is the the natural convengence of the grid services, as defined in OGSA, and the Web services framework.

In the following sections, we present basic specifications for WSRF. Specification authors plan to submit them to an appropiate standards body in the near future. These drafts have already been made available to the GGF OGSI working group for comments.

WS-ResourceLifetime

This defines the mechanisms for WS-Resource destruction, including message exchanges that allow a requestor to destroy a resource, either immediately or by using a time-based scheduled resource termination mechanism.

WS-ResourceProperties

Defines how the type of definition of a WS-Resource can be associated with the interface description of a Web service, and message exchanges for retrieving, changing, and deleting WS-Resource properties. This relationship is the implied resource pattern.

WS-Notification

Defines mechanisms for event subscription and notification using a topic-based publish/subscribe pattern.

WS-RenewableReferences

Defines a conventional decoration of a WS-Addressing endpoint reference with policy information needed to retrieve an updated version of an endpoint reference when it becomes invalid.

WS-ServiceGroup

Defines an interface to heterogeneous by-reference collections of Web services. ServicesGroup can be used to form a wide variety of collections of services or resources, including registries of services and associated resources.

WS-BaseFault

Defines an XML Schema type for base faults, along with rules for how this base fault type is used and extended by Web services; this simplifies problem determination by standardizing a base set of information that would appear in fault messages.

WS-Resource Framework: some definitions

In the following sections, some definitions are included regarding WSRF. This definitions are extracted from documents still work in progress. Please refer to the pointers at the end of this appendix for further updates.

Web service

The term Web services emerged in the year 2000 with the introduction of technologies such as SOAP (Simple Object Access Protocol), WSDL (Web Services Description Language) and UDDI (Universal Description Discovery and Integration). Later, the term SOA (Service Oriented Architecture) was coined to describe the overall approach of building loosely coupled distributed systems with minimal shared understanding among system components.

Web services are basically Web-based applications, but they are different in the sense that they are designed to support application to application communication.

Definition: A Web service is a software system designed to support interoperable machine-to-machine interaction over a network.

A Web service is a component deployed within some runtime environment which is responsible for executing the code of the Web service and for dispatching messages to the Web service. IBM's WebSphere and JBoss are two examples of runtime environments.

A Web service, as defined in the WSRF, is stateless, meaning that it is a service whose implementation *maintains no dynamic data,* but which acts upon stateful resources (documents) based on messages it sends and receives. When a Web service is stateless, it will not maintain a dynamic state, meaning a state for which the service is responsible between message exchanges with its requestors. This also brings some advantages; a stateless Web service can be restarted following failure without concern for its history or prior interactions, and more copies can be created or destroyed in response to the changing load.

A Web service should be designed while keeping in mind the Web service interface. This Web service interface is described by using WSDL, *and defines the Web service capabilities in terms of a collection of operations that may be*

invoked by other entities, known as service requestors. Any other Web service can be at some point in time a service requestor.

WS-Resource

If Web services are supposed to be stateless, or considered as a stateless message processor, message exchanges are, in many cases, supposed to enable access/update to state maintained by other system components; those file systems, databases or other entities, can also be considered to be stateful resources. The link between one or more stateful resource and a Web service is the Implied Resource Pattern. The Implied Resource Pattern is a *set of conventions on Web services technologies, in particular XML, WSDL and WS-Addressing,* implicitly meaning that the requestor does not provide the identity if the resource has an explicit parameter in the body of the request message. The context used to designate the implied stateful resource is encapsulated in the WS-Addressing endpoint reference used to address the target Web service at its endpoint. The term *pattern* indicates that the relationship between Web services and stateful resources is codified by a set of conventions on existing Web services technologies, XML, WSDL, and WS-Addressing. Please note that when the Web service itself is stateless and participating with these other two elements, the result can be a stateful operation.

WS-Resource is the result of the participation of a stateful resource in the implied resource pattern.

Related publications

The publications listed in this section are considered particularly suitable for a more detailed discussion of the topics covered in this redbook.

IBM Redbooks

For information on ordering these publications, see “How to get IBM Redbooks” on page 270. Note that some of the documents referenced here may be available in softcopy only.

- *Fundamentals of Grid Computing,* REDP-3613-00
- *Introduction to Grid Computing with Globus,* SG24-6895-01
- *Enabling Applications for Grid Computing with Globus,* SG24-6936-00
- *Globus Toolkit 3.0 Quick Start*, REDP-3697-00
- *Using a callback mechanism with Globus*, TIPS0190
- *How to Organize a Localization Pack*, TIPS0130
- *Linux Handbook - A Guide to IBM Linux Solutions and Resources*, SG24-7000-00
- *e-business On Demand Operating Environment*, REDP-3673-00

Other publications

These publications are also relevant as further information sources:

- Foster, et al, *The Grid: Blueprint for a New Computing Infrastructure,* Morgan Kaufmann, 1999, ISBN 1558604758
- *The Anatomy of the Grid: Enabling Scalable Virtual Organizations*, at:

  http://www.globus.org/research/papers/anatomy.pdf
- *Web Services Conceptual Architecture*, 2001, by Heather Kreger - IBM Software Group
- *Globus Java Programmer’s Guide Core Framework*, at:

  http://www-unix.globus.org/toolkit/3.0/ogsa/docs/java_programmers_guide.html
- *Grid Service Development Tools Guide*, at:

  http://www-unix.globus.org/toolkit/3.0/ogsa/docs/tools_guide.html

- *The Globus Toolkit 3 Programmer's Tutorial*, found at:

  http://www.casa-sotomayor.net/gt3-tutorial/
- *Design an application for grid*, November 2003, IBM developerWorks > Grid Computing
- *How to build a Grid Service using GT3*, at:

  http://www-unix.mcs.anl.gov/~bacon/tutorial/
- *WS-Resource Framework Documents*, at:

  http://www.globus.org/wsrf/#relevant

Online resources

These Web sites and URLs are also relevant as further information sources:

Include references to Use Cases (description, tools)

- Globus Alliance

  http://www.globus.org
- Global Grid Forum (GGF)

  http://www.ggf.org/
- GridForge - working respository for GGF Working and Research Groups

  http://forge.gridforum.org/
- OASIS Technical Committee

  http://www.oasis-open.org/
- GlobusWORLD

  http://www.globusworld.org/
- Grid developerWorks Web site

  http://www.ibm.com/developerworks/grid
- OGSI Version 1.0 (Draft)

  http://www.gridforum.org/ogsi-wg/drafts/draft-ggf-ogsi-gridservice-29_2003-04-05.pdf
- Grid Service Specification (Draft 3)

  http://www.globus.org/research/papers/gsspec.pdf
- Apache Software Foundation

  http://www.apache.org

- Apache Ant Project

  http://ant.apache.org
- Apache Axis Project

  http://ws.apache.org/axis/
- Apache Jakarta Project

  http://jakarta.apache.org
- Apache Software license

  http://www.opensource.org/licenses/apachepl.php
- Condor

  http://www.cs.wisc.edu/condor/
- World Wide Web Consortium (W3C)

  http://www.w3.org/
- SOAP 1.1 specification

  http://www.w3.org/TR/SOAP/
- WSDL 1.1 specification

  http://www.w3.org/TR/wsdl
- Web Service Description Language (WSDL)

  http://www.w3.org/
- Open source Initiative license information

  http://opensource.org/docs/certification_mark.php
- Lesser General Public License

  http://www.opensource.org/licenses/lgpl-license.php
- GNU Public License

  http://www.gnu.org/copyleft/gpl.html
- IBM LoadLeveler®

  http://www.ibm.com/servers/eserver/pseries/library/sp_books/loadleveler.html
- IBM Public License

  http://www.opensource.org/licenses/ibmpl.php
- IBM eServer Information Center

  http://publib.boulder.ibm.com/eserver/v1r1/en_US/index.html?info/ogsainfo/kickoff.html
- Grid Computing Environment (GCE)

  http://www.globus.org/research/development-environments.html

- IBM Grid Toolbox

  http://www.alphaworks.ibm.com/tech/gridtoolbox
- Grid Application Framework for Java

  http://www.alphaworks.ibm.com/tech/GAF4J
- Globus Java Programmer's Guide Core Framework at:

  http://www-unix.globus.org/toolkit/3.0/ogsa/docs/java_programmers_guide.html
- Grid Service Development Tools Guide at:

  http://www-unix.globus.org/toolkit/3.0/ogsa/docs/tools_guide.html
- The Globus Toolkit 3.0 Programmer's Tutorial

  http://www.casa-sotomayor.net/gt3-tutorial/
- Platform JobScheduler 5

  http://www.platform.com/
- Portable Batch System (PBS)

  http://pbs.mrj.com/
- OpenPegasus

  http://www.openpegasus.org/
- Distributed Management Task Force (DMTF)

  http://www.dmtf.org/about
- Standards Based Linux Instrumentation for Manageability (SBLIM)

  http://www.ibm.com/sblim

How to get IBM Redbooks

You can search for, view, or download Redbooks, Redpapers, Hints and Tips, draft publications and Additional materials, as well as order hardcopy Redbooks or CD-ROMs, at this Web site:

ibm.com/redbooks

Help from IBM

IBM Support and downloads:

ibm.com/support

IBM Global Services:

ibm.com/services

Index

E

F

G

H

I

J

L

M

N

O

T

U

V

W

X

Grid Services Programming and Application Enablement

(0.5” spine)
0.475”<->0.875”
250 <-> 459 pages

Grid Services Programming and Application Enablement

Grid services programming

Samples using Globus Toolkit V3.0

Implementation based on OGSI V1.0

The goal of this IBM Redbook is to familiarize the user with the concepts of the OGSA (Open Grid Services Architecture), OGSI (Open Grid Services Infrastructure), Globus Toolkit V3.0, presenting concrete programmatic examples, and also introducing the enhanced features of the IBM Grid Toolbox.

We illustrate the various steps needed to develop a grid service application. Existing applications can be wrappered and made available as grid services or applications can be developed from scratch to take advantage of the grid service concepts and provide the benefits made possible by that grid service.

INTERNATIONAL TECHNICAL SUPPORT ORGANIZATION

BUILDING TECHNICAL INFORMATION BASED ON PRACTICAL EXPERIENCE

IBM Redbooks are developed by the IBM International Technical Support Organization. Experts from IBM, Customers and Partners from around the world create timely technical information based on realistic scenarios. Specific recommendations are provided to help you implement IT solutions more effectively in your environment.

For more information:
ibm.com/redbooks

SG24-6100-00 ISBN 0738498033