The Globus Toolkit 4 Programmer's Tutorial

Borja Sotomayor
University of Chicago
Department of Computer Science

The Globus Toolkit 4 Programmer's Tutorial

by Borja Sotomayor

Revision History

Revision 0.2.1 26 Nov 2005 Revised by: Borja Sotomayor
Added placeholder chapters for Information Services, Execution Management, and Data Management, to emphasize that the tutorial wi
Revision 0.2 15 Oct 2005 Revised by: Borja Sotomayor
Added security chapters. The first two security chapters are very similar to the ones found in The GT3 Programmer's Tutorial (an introd
Revision 0.1.1 19 Jun 2005 Revised by: Borja Sotomayor
Updated to GSBT (http://gsbt.sourceforge.net/) 0.2.4 which fixes an important bug in the Python build script.
Revision 0.1 13 May 2005 Revised by: Borja Sotomayor
First stable release of the tutorial. The examples have been test
Revision 0.0.7.1 21 Apr 2005 Revised by: Borja Sotomayor
Fixed buildfile + script so it won't fail if it does not find an "etc" directory.
Revision 0.0.7 13 Apr 2005 Revised by: Borja Sotomayor
Examples have been tested with GT3.9.5. The tutorial examples now implement the singleton or the factory/instance p
Revision 0.0.6 08 Feb 2005 Revised by: Borja Sotomayor
The tutorial now includes (in the Additional Examples appendix) two examples (a Singleton and a Factory example) that more accurate
Revision 0.0.5 29 Dec 2004 Revised by: Borja Sotomayor
First 'readable' version of the First WSRF Web Service chapter (instead of just a collection of notes). Added "I
Revision 0.0.4 23 Dec 2004 Revised by: Borja Sotomayor
First 'readable' version of the Key Concepts chapter (instead of just a collection of notes).
Revision 0.0.3 19 Dec 2004 Revised by: Borja Sotomayor
Examples work with GT3.9.4 Polished source code of example (uses ReflectionResourceProperty to make code a bit clea
Revision 0.0.2 13 Nov 2004 Revised by: Borja Sotomayor
Polished source code of example. Included new (easier to use) b
Revision 0.0.1 06 Nov 2004 Revised by: Borja Sotomayor
Very first version of the GT4 Tutorial.

Table of Contents

List of Tables

Introduction

Welcome to the Globus Toolkit 4 Programmer's Tutorial! This document is intended as a starting point for anyone who is going to develop applications using the Globus Toolkit 4 (GT4).

The tutorial is divided into 3 main areas:

- **Getting Started**: An introduction to key concepts related with GT4 and the Web Services Resource Framework (WSRF)
- **GT4 Java Core**: A guide to programming basic Web Services which only use the Java WS Core component of GT4.
- **GT4 Security**: A guide to programming *secure* Web Services using the toolkit's security components.

Please note that the tutorial, at this point, only covers the *Java WS Core* component and a small part of the security components of the toolkit. These are an important but *small* part of the whole toolkit. At the end of this tutorial you will know how to program stateful Web services using GT4. This will allow you to progress towards using the higher-level services of the toolkit (using official Globus documentation). However it is important to understand that *you cannot program Grid-based applications using only the Java WS Core component of the toolkit*. This tutorial should be approached as a stepping stone towards more powerful tooling, not as a definite guide on Grid programming.

GT4 Prerequisite Documents

This tutorial has no GT4 prerequisite documents, since it is intended as a starting point for GT4 programmers. However, you should already be familiar with Grid Computing. A good, short introduction to what Grid Computing is can be found in Ian Foster's paper The Grid: A New Infrastructure for 21st Century Science (http://www.aip.org/pt/vol-55/iss-2/p42.html). A more extense, and very easy to read, introduction can be found at the Grid Café (http://gridcafe.org/).

For a much more detailed text, you might want to check out the following book: The Grid 2: Blueprint for a New Computing Infrastructure (http://www.mkp.com/grid2) (Edited by Ian Foster and Carl Kesselman. 2003). Most of the book is easy to read and not too technical. It is also known as "The Grid Bible". With a name like that, you can assume it's worth taking a look at it :-)

You might also be interested in taking a look at the 'Publications' section in the Globus website (http://www.globus.org), specially the documents listed below. However, these documents are rather technical and might be too hard for a beginner. You might want to just skim through them at first, and then reread them once you're familiar with GT4.

- The Anatomy of the Grid: Enabling Scalable Virtual Organizations (http://www.globus.org/research/papers/anatomy.pdf) . I. Foster, C. Kesselman, S. Tuecke.
- The Physiology of the Grid: An Open Grid Services Architecture for Distributed Systems Integration (http://www.globus.org/research/papers/ogsa.pdf) . I. Foster, C. Kesselman, J. Nick, S. Tuecke.

Audience

This document is intended for programmers who are new to the Globus Toolkit 4 (GT4).

Assumptions

The following knowledge is assumed:

- Programming in Java. If you don't know Java, you can find some useful links . Also, prior experience of distributed systems programming with Java (with CORBA, RMI, etc.) will certainly come in handy, but is not strictly required.
- Basic knowledge of XML. If you have no idea of XML, you can find some useful links .
- You should know your way around a UNIX system. This tutorial is mainly UNIX-oriented, although in the future we hope to include sections for Windows users.
- Basic knowledge of what the Grid and grid-based applications are. This tutorial is not intended as an introduction to Grid Computing, but rather as an introduction to a toolkit which can enable you to program grid-based applications.

The following knowledge is *not* required:

- Web Services. The tutorial includes an introduction to fundamental Web Services concepts needed to use GT4.
- Globus Toolkit 2 or 3

Related Documents

- Specifications
  - OASIS WSRF Page (http://www.oasis-open.org/committees/tc_home.php?wg_abbrev=wsrf): The most recent version of the WSRF specifications can be found here.

    Globus WSRF Page (http://www.globus.org/wsrf/): Contains pointers to papers and presentations related to WSRF.
- Official Globus Documentation
  - GT4 Fact Sheet (http://www-unix.globus.org/toolkit/GT4Facts/): Everything there is to know about GT4.
  - Official GT4.0 Documentation (http://www.globus.org/toolkit/docs/4.0/): Includes the installation guide.
- Other related documents

- Ian Foster's Globus Toolkit Primer (http://www.globus.org/toolkit/docs/4.0/key/GT4_Primer_0.6.pdf) provides a detailed look at the toolkit and all the components included in it.
- Scott Gose's WSRF Grid Services (http://www-unix.mcs.anl.gov/%7Egose/grid-services/) page. Includes easy-to-understand instructions on how to set up a minimal WSRF container.

Document Conventions

The following conventions will be observed throughout this document:

Code

```
public class HelloWorld
{
  public static final void main( String args[] )
  {
     ❶
     // Code in bold is important
     System.out.println("Hello World");
  }
}
```

❶ This is a callout, further explaining an important part of the code.

Inlined code

Whenever we refer to bits of code from the main text, it will be highlighted `like this`. For example:

> The `HelloWorld` class has a single `main` method that prints out a `"Hello World"` string.

Shell commands

```
javac HelloWorld.java
```

If a command is too long to fit in a single line, it will be wrapped into several lines using the backslash ("\") character. On most UNIX shells (including BASH) you should be able to copy and paste all the lines at once into your console.

```
javac \
-classpath /usr/lib/java/Hello.jar \
HelloWorld.java \
HelloUniverse.java \
HelloEveryone.java
```

Notes

You can find three types of notes in the text: complementary information, reminders, and warnings.

Tip: This is a complementary information block.

This kind of note contains interesting information that complements what is currently being discussed in the text.

Note: This is a reminder.

This kind of notes are usually used after a block of code to remind you of where you can find the file that contains that particular code. It is also used to remind you of important concepts, and to suggest what sections of the tutorial you should read again if you have a hard time understanding a particular section.

Caution

This is a warning.

Warnings are used to emphatically point out something. They generally refer to common pitfalls or to things that you should take into account when writing your own code.

About the author & acknowledgments

The Globus Toolkit 4 Programmer's Tutorial is written and maintained by Borja Sotomayor, a Ph.D. student at the Department of Computer Science (http://www.cs.uchicago.edu/) at the University of Chicago (http://www.uchicago.edu/). You can find out more about me in my UofC personal page (http://people.cs.uchicago.edu/~borja/).

Acknowledgments

This tutorial can hardly be considered a one-person effort. The following people have, in one way or another, helped to make the GT4 Tutorial a reality:

- Lisa Childers
- Rebeca Cortazar (http://paginaspersonales.deusto.es/cortazar/)

- Ian Foster (http://www-fp.mcs.anl.gov/~foster/)
- Leon Kuntz (in memoriam)
- Jesus Marco
- All the Globus gurus who have reviewed the tutorial on countless occasions

A lot of readers have helped to improve the tutorial by reporting bugs and typos, as well as making very constructive comments and suggestions:

> Rodrigo Calheiros, Nicholas Chase, Ben Clifford, Nurcan Coskun, Joe Davey, Andreas Dinges, Kay Dörnemann, Felipe Franciosi, Tim Freeman, Pedro Gama, Scott Gose, Ming Jiang, Makoto Kisimoto, Martin Kuba, Sam Meder, Alexandre Prado Teles, V.Shirikov, Frank Siebenlist, Larry Tan, Johan Tordsson, Thomas Weishäupl

If you've reported a bug, typo, or helped out in any way, and you are not listed here, please do let me know!

The following people helped out back when the GT4 Tutorial was the GT*3* Tutorial:

> Balamurali Ananthan, Sebastien Barre, Thomas Becker, Luther Blake, Robert M. Bram, Javier Cano, Paulo Cortes, Jun Ebihara, Qin Feng, Luis Ferreira, Fernando Fraticelli, Anders Keldsen, Britt Johnston, Steve Mock, Elizabeth Post, Philippe Prados, Michael Schneider, Shiva Shankar Chetan, Nelson Sproul, Ian Stokes-Rees, Jason Young, Matthew Vranicar, James Werner

I. Getting Started

Chapter 1. Key Concepts

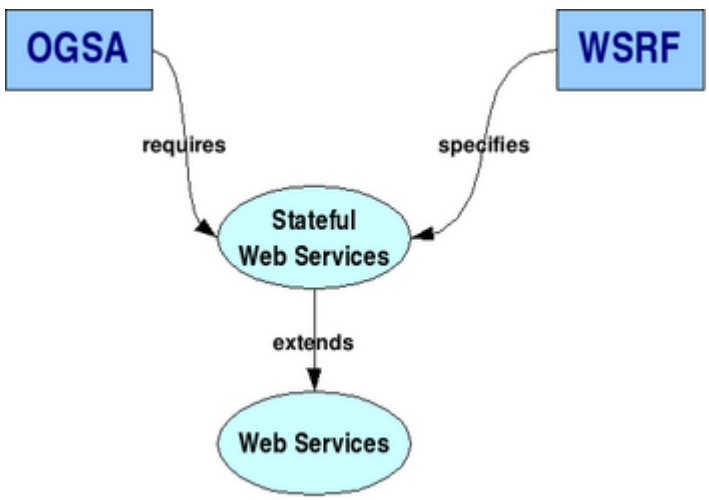
There are certain key concepts that must be well understood before being able to program with GT4. This chapter gives a brief overview of all those fundamental concepts.

- **OGSA, WSRF, and GT4**: We'll take a look at what these oft-mentioned acronyms mean, and how they are related.
- **Web Services**: OGSA, WSRF, and GT4 are based on standard Web Services technologies such as SOAP and WSDL. You don't need to be a Web Services expert to program with GT4, but you should be familiar with the Web Services architecture and languages. We provide a basic introduction and give you pointers to interesting sites about Web Services.
- **The Web Services Resource Framework**: WSRF is the core of GT4. We take a look at what a Web Service Resource (or WS-Resource) is, and how it is related to Web Services.
- **The GT4 Architecture**: After seeing both WS-Resources and Web Services, we take a look at the whole GT4 architecture, and how WSRF fits in it.
- **Java & XML**: Finally, if you want to use GT4, you need to be able to program in Java, and to understand basic XML. If you're new to Java and XML, we provide a couple links that can help you get started.

OGSA, WSRF, and GT4

If you've started looking at GT4, you've probably encountered at least these two acronyms: OGSA and WSRF. But... what do they mean? How are they related to the Globus Toolkit 4? This section attempts to clarify these important concepts. First of all, let's start by taking a look at OGSA and WSRF *without* seeing just yet how they are related to GT4.

Figure 1-1. Relationship between OGSA, WSRF, and Web Services



OGSA

A grid application will usually consist of several different components. For example, a typical grid application could have:

- **VO Management Service**: To manage what nodes and users are part of each Virtual Organization.
- **Resource Discovery and Management Service:** So applications on the grid can discover resources that suit their needs, and then manage them.
- **Job Management Service**: So users can submit tasks (in the form of "jobs") to the Grid.
- And a whole other bunch of services like security, data management, etc.

> **Note:** If you have absolutely no idea of what I have just said (and feel slightly confused), remember that the tutorial assumes that you know what Grid Computing is. If you don't, please read the Prerequisites documents section to get up to speed.

Furthermore, all these services are interacting constantly. For example, the Job Management Service might consult the Resource Discovery Service to find computational resources that match the job's requirements. With so many services, and so many interactions between them, there exists the potential for chaos. What if every vendor out there decided to implement a Job Management Service in a completely different way, exposing not only different functionality but also different interfaces? It would be very difficult (or nearly impossible) to get all the different software pieces to work together.

The solution is *standardization*: define a common interface for each type of service. For example, take a look at the World Wide Web. One of the reasons why the Web is such a popular Internet application is because it is based on *standards* (HTML, HTTP, etc.) agreed upon by all the different major players (Microsoft, Netscape, etc.). Imagine, on the other hand, that you could only use a Microsoft browser to access websites implemented with Microsoft technology (ditto for Netscape, Opera, etc.) It would be definitely uncool. Thanks to standards, I can use my favorite browser (provided it follows standards, which most modern browsers do) to access most of the websites out there (regardless of what technology is used to implement the website). Why? Because a set of common languages was agreed upon for all the browsers and websites out there. Standardization is definitely a good thing.

The Open Grid Services Architecture (OGSA), developed by The Global Grid Forum (http://www.ggf.org), aims to define a common, standard, and open architecture for grid-based applications. The goal of OGSA is to standardize practically all the services one commonly finds in a grid application (job management services, resource management services, security services, etc.) by specifying a set of standard interfaces for these services. At the time of writing this tutorial, this "set of standard interfaces" is still in the works. However, OGSA already defines a set of requirements that must be met by these standard interfaces. In other words, OGSA has already gone as far as identifying the most important services one encounters in Grid applications, and which most stand to benefit from standardization.

OGSA requires ’stateful services’

However, when the powers-that-be undertook the task of creating this new architecture, they realized they needed to choose some sort of distributed middleware on which to *base* the architecture. In other words, if OGSA (for example) defines that the JobSubmissionInterface has a submitJob method, there has to be a common and standard way to *invoke* that method if we want the architecture to be adopted as an industry-wide standard. This *base* for the architecture could, in theory, be any distributed middleware (CORBA, RMI, or even traditional RPC). For reasons that will be explained further on, Web Services were chosen as the underlying technology.

However, although the Web Services Architecture was certainly the best option, it still didn’t meet one of OGSA’s most important requirements: the underlying middleware had to be *stateful* (don’t worry if you don’t know what a "stateful" service is, it is explained in the next section). Unfortunately, although Web services can in theory be either stateless or stateful, they are usually stateless and there is no standard way of making them stateful. So, clearly, something had to be done!

WSRF

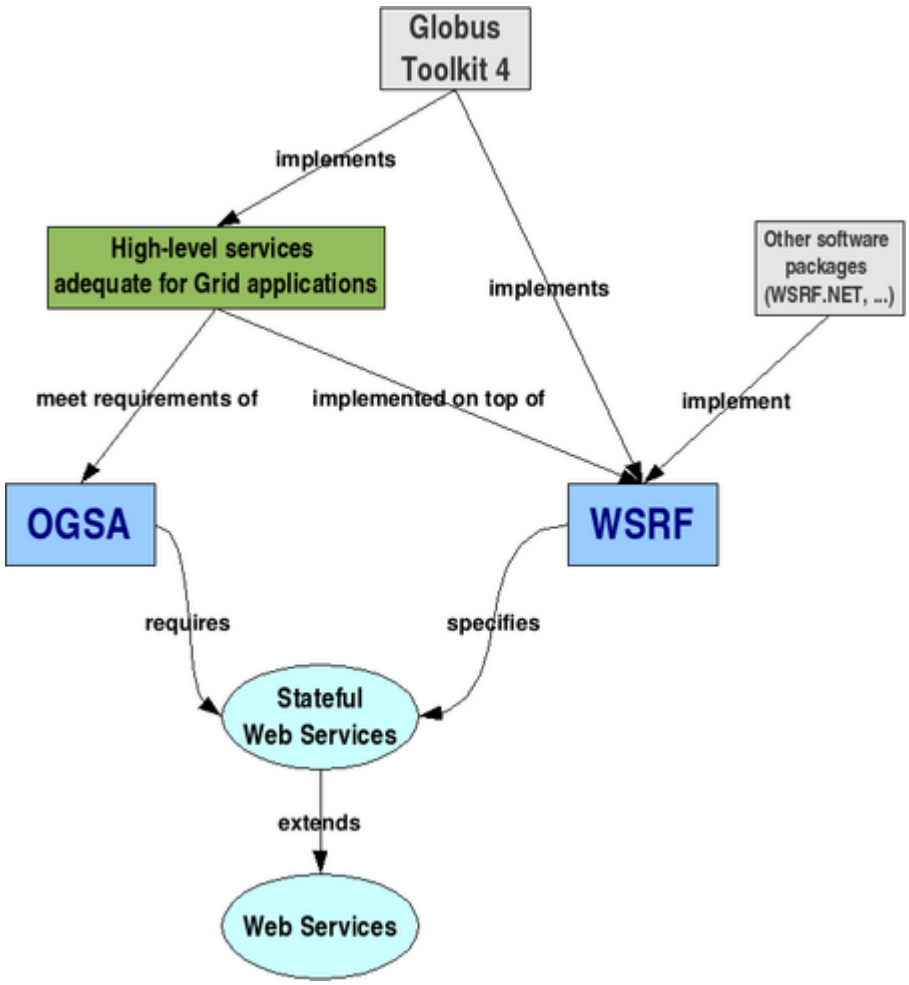
Enter the Web Services Resource Framework, a specification developed by OASIS (http://www.oasis-open.org). WSRF specifies how we can make our Web Services stateful, along with adding a lot of other cool features. It is important to note that WSRF is a joint effort by the Grid and Web Services communities, so it fits pretty nicely inside the whole Web Services Architecture (in the diagram: *WSRF extends Web Services*).

So what exactly is the relation between OGSA and WSRF? It’s very simple: WSRF provides the stateful services that OGSA needs. In the diagram: **WSRF specifies stateful services** (as opposed to those services simply ’being required’ by OGSA). Another way of expressing this relation is that, while OGSA is the *architecture*, WSRF is the *infrastructure* on which that architecture is built on.

How does this relate to GT4?

Now that we’ve cleared up what OGSA and WSRF are, we are ready to complete the above diagram to see how GT4 fits into the picture:

Figure 1-2. Relationship between OGSA, GT4, WSRF, and Web Services



The Globus Toolkit 4

The Globus Toolkit is a software toolkit, developed by The Globus Alliance (http://www.globus.org), which we can use to program grid-based applications. The toolkit, first and foremost, includes quite a few *high-level services* that we can use to build Grid applications. These services, in fact, meet most of the abstract requirements set forth in OGSA. In other words, the Globus Toolkit includes a resource monitoring and discovery service, a job submission infrastructure, a security infrastructure, and data management services (to name a few!). Since the working groups at GGF are still working on defining standard interfaces for these types of services, we can't say (at this point) that GT4 is an implementation of OGSA (although GT4 does implement some security specifications defined by GGF). However, it *is* a realization of the OGSA requirements and a sort of *de facto* standard for the Grid community while GGF works on standardizing all the different services.

Most of these services are implemented *on top of WSRF* (the toolkit also includes some services that are not implemented on top of WSRF and are called the *non-WS components*). The Globus Toolkit 4, in fact, includes a complete implementation of the WSRF specification. This part of the toolkit (the WSRF

implementation) is a very important part of the toolkit since nearly everything else is built on top of it. However, it's worth noting that it's also a *very small* part of the toolkit. At this point, we'll repeat something we said at the very beginning of the tutorial:

> At the end of this tutorial you will know how to program stateful Web Services using GT4. This will allow you to progress towards using the higher-level services of the toolkit. However it is important to understand that *you cannot program Grid-based applications using only the Java WS Core included in this tutorial*. This tutorial should be approached as a stepping stone towards more powerful tooling, not as a definite guide on GT4 programming.

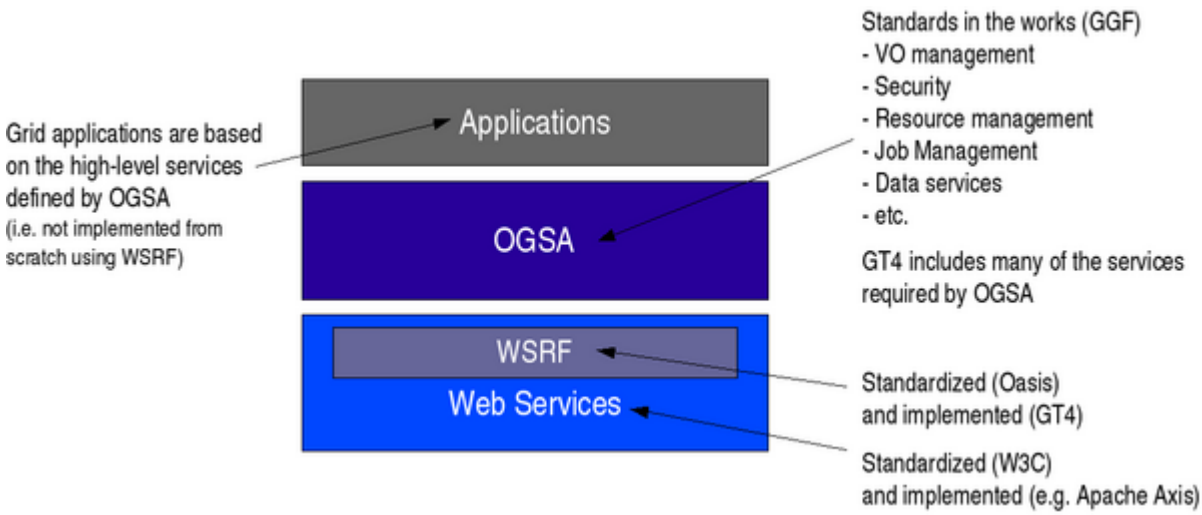
I'm sorry to insist so much on this, but this really is a very important concept. Never forget that, on the long road towards true Grid Nirvana, WSRF is indeed a necessary step, but only the *first* step.

Finally, take into account that GT4 isn't the only WSRF implementation out there. For example, another complete implementation of the WSRF specification is WSRF.NET (http://www.cs.virginia.edu/~gsw2c/wsrf.net.html).

The mandatory layered diagram

If there's something we computer geeks like in documentation, it's layered diagrams! So, this section really wouldn't be complete without a diagram that explains the relationship between OGSA, WSRF, and GT4 using the wonderful language of layers.

Figure 1-3. Layered diagram of OGSA, GT4, WSRF, and Web Services



A short introduction to Web Services

Before we take a closer look at what the Web Services Resource Framework (WSRF) is, we need to have a basic understanding of how Web Services work (so we can better appreciate how WSRF extends Web Services). If you're already familiar with Web Services, you can safely skip this section.

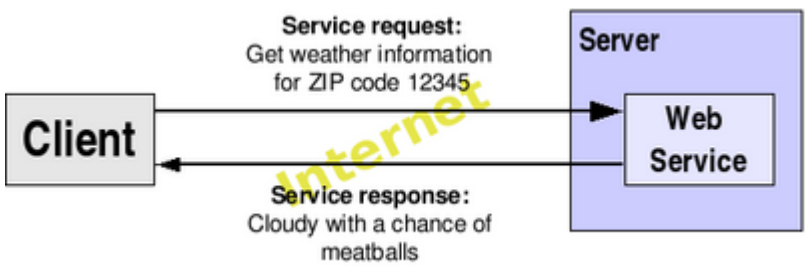
For quite a while now, there has been a lot of buzz about "Web Services," and many companies have begun to rely on them for their enterprise applications. So, what exactly are Web Services? To put it quite simply, they are *yet another* distributed computing technology (like CORBA, RMI, EJB, etc.). They allow us to create client/server applications.

For example, let's suppose I keep a database with up-to-date information about weather in the United States, and I want to distribute that information to anyone in the world. To do so, I could *publish* the weather information through a Web Service that, given a ZIP code, will provide the weather information for that ZIP code.

> **Caution**
>
> Don't mistake this with publishing something on a *website*. Information on a website (like the one you're reading right now) is intended for humans. Information which is available through a Web Service will *always* be accessed by software, *never* directly by a human (despite the fact that there might be a human using that software). Even though Web Services rely heavily on existing Web technologies (such as HTTP, as we will see in a moment), they have no relation to web browsers and HTML. Repeat after me: websites for humans, Web Services for software :-)

The *clients* (programs that want to access the weather information) would then contact the *Web Service* (in the *server*), and send a *service request* asking for the weather information. The server would return the forecast through a *service response*. Of course, this is a very sketchy example of how a Web Service works. We'll see all the details in a moment.

Figure 1-4. Web Services



Some of you might be thinking: *"Hey! Wait a moment! I can do that with RMI, CORBA, EJBs, and countless other technologies!"* So, what makes Web Services special? Well, Web Services have certain advantages over other technologies:

- Web Services are platform-independent and language-independent, since they use standard XML languages. This means that my client program can be programmed in C++ and running under Windows, while the Web Service is programmed in Java and running under Linux.
- Most Web Services use HTTP for transmitting messages (such as the service request and response). This is a major advantage if you want to build an Internet-scale application, since most of the Internet's proxies and firewalls won't mess with HTTP traffic (unlike CORBA, which usually has trouble with firewalls).

Of course, Web Services also have some disadvantages:

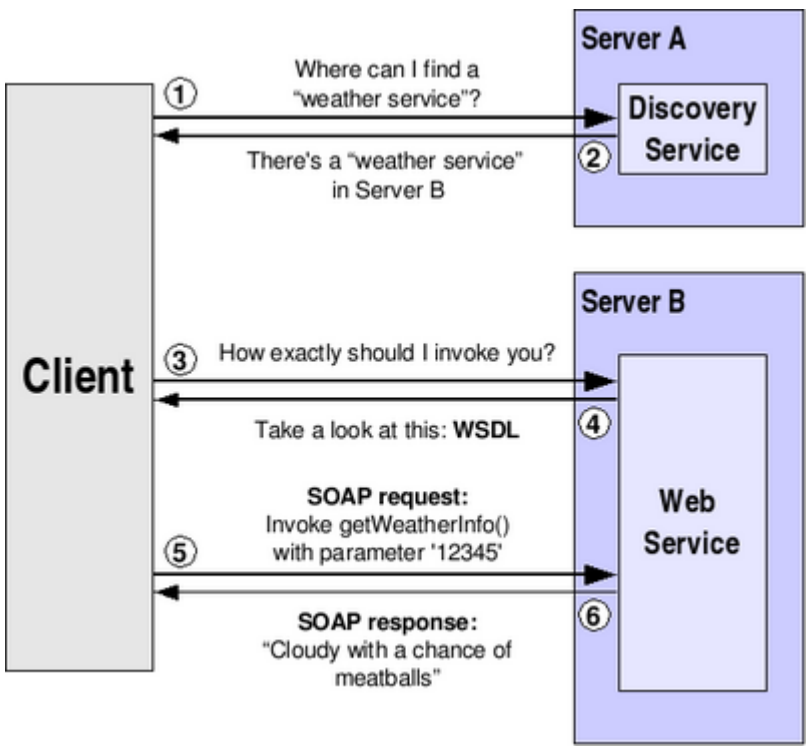
- Overhead. Transmitting all your data in XML is obviously not as efficient as using a proprietary binary code. What you win in portability, you lose in efficiency. Even so, this overhead is usually acceptable for most applications, but you will probably never find a critical real-time application that uses Web Services.
- Lack of versatility. Currently, Web Services are not very versatile, since they only allow for some very basic forms of service invocation. CORBA, for example, offers programmers a lot of supporting services (such as persistency, notifications, lifecycle management, transactions, etc.). Fortunately, there are a lot of emerging Web services specifications (including WSRF) that are helping to make Web services more and more versatile.

However, there is one important characteristic that distinguishes Web Services. While technologies such as CORBA and EJB are geared towards *highly coupled* distributed systems, where the client and the server are very dependent on each other, Web Services are more adequate for *loosely coupled* systems, where the client might have no prior knowledge of the Web Service until it actually invokes it. Highly coupled systems are ideal for intranet applications, but perform poorly on an Internet scale. Web Services, however, are better suited to meet the demands of an Internet-wide application, such as grid-oriented applications.

A Typical Web Service Invocation

So how does this all actually work? Let's take a look at all the steps involved in a complete Web Service invocation. For now, don't worry about all the acronyms (SOAP, WSDL, ...). We'll explain them in detail in just a moment.

Figure 1-5. A typical Web Service invocation

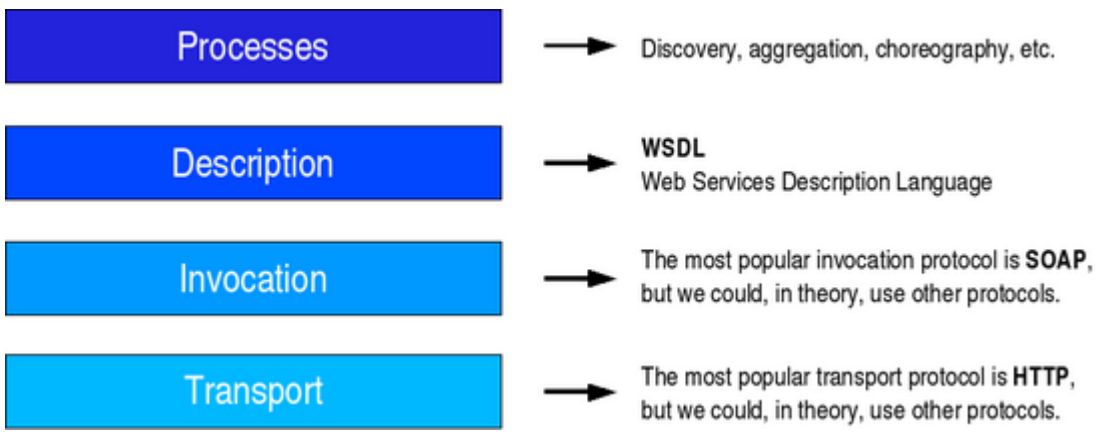


1. As we said before, a client may have no knowledge of what Web Service it is going to invoke. So, our first step will be to *discover* a Web Service that meets our requirements. For example, we might be interested in locating a public Web Service which can give me the weather forecast in US cities. We'll do this by contacting a *discovery service* (which is itself a Web service).
2. The discovery service will reply, telling us what servers can provide us the service we require.
3. We now know the location of a Web Service, but we have no idea of how to actually invoke it. Sure, we know it can give me the forecast for a US city, but how do we perform the actual service invocation? The method I have to invoke might be called "`string getCityForecast(int CityPostalCode)`", but it could also be called "`string getUSCityWeather(string cityName, bool isFarenheit)`". We have to ask the Web Service to *describe* itself (i.e. tell us how exactly we should invoke it)
4. The Web Service replies in a language called WSDL.
5. We finally know where the Web Service is located and how to invoke it. The invocation itself is done in a language called SOAP. Therefore, we will first send a *SOAP request* asking for the weather forecast of a certain city.
6. The Web Service will kindly reply with a *SOAP response* which includes the forecast we asked for, or maybe an error message if our SOAP request was incorrect.

Web Services Architecture

So, what exactly are SOAP and WSDL? They're essential parts of the Web Services Architecture:

Figure 1-6. The Web Services architecture



- **Service Processes**: This part of the architecture generally involves more than one Web service. For example, discovery belongs in this part of the architecture, since it allows us to locate one particular service from among a collection of Web services.
- **Service Description**: One of the most interesting features of Web Services is that they are *self-describing.* This means that, once you've located a Web Service, you can ask it to 'describe itself'

and tell you what operations it supports and how to invoke it. This is handled by the Web Services Description Language (WSDL).

- **Service Invocation**: Invoking a Web Service (and, in general, any kind of distributed service such as a CORBA object or an Enterprise Java Bean) involves passing messages between the client and the server. SOAP (Simple Object Access Protocol) specifies how we should format requests to the server, and how the server should format its responses. In theory, we could use other service invocation languages (such as XML-RPC, or even some *ad hoc* XML language). However, SOAP is by far the most popular choice for Web Services.
- **Transport**: Finally, all these messages must be transmitted somehow between the server and the client. The protocol of choice for this part of the architecture is HTTP (HyperText Transfer Protocol), the same protocol used to access conventional web pages on the Internet. Again, in theory we could be able to use other protocols, but HTTP is currently the most used one.

In case you're wondering, most of the Web Services Architecture is specified and standardized by the World Wide Web Consortium (http://www.w3c.org/), the same organization responsible for XML, HTML, CSS, etc.

Web Services Addressing

We have just seen a simple Web Service invocation. At one point, a discovery service 'told' the client *where* the Web Service is located. But... how exactly are Web services addressed? The answer is very simple: just like web pages. We use plain and simple URIs (Uniform Resource Identifiers). If you're more familiar with the term URL (Uniform Resource Locator), don't worry: URI and URL are practically the same thing.

For example, the discovery registry might have replied with the following URI:

```
http://webservices.mysite.com/weather/us/WeatherService
```

This could easily be the address of a web page. However, remember that Web Services are always used by software (never directly by humans). If you typed a Web Service URI into your web browser, you would probably get an error message or some unintelligible code (some web servers *will* show you a nice graphical interface to the Web Service, but that isn't very common). When you have a Web Service URI, you will usually need to give that URI to a program. In fact, most of the client programs we will write will expect to receive a Web service URI as a command-line argument.

> **Tip:** If you're anxious to see a real Web service working, then today's your lucky day! A "Weather Web Service" is probably one of the most typical examples of a simple web service. You can find a real Weather Web Service here:
>
> http://live.capescience.com/ccx/GlobalWeather
>
> Wait a second... You didn't actually try to visit that URI, did you? Haven't you been paying attention? That's a Web service URI, so even though it may look and feel like the URIs you type in your browser when you want to visit your favorite website, this URI is meant only for software that "knows" how to invoke Web services.
>
> Fortunately, the authors of that web service have been kind enough to provide a description (http://www.capescience.com/webservices/globalweather/index.shtml) of the Web service, along with

a web interface (http://live.capescience.com/GlobalWeather) so you can actually invoke the service's methods. If you feel specially curious, you can even take a look at the Web service's WSDL (http://live.capescience.com/wsdl/GlobalWeather.wsdl) (also available in a slightly more readable version (http://www.w3.org/2000/06/webdata/xslt?xslfile=http://www.capescience.com/simplifiedwsdl.xslt&xmlfile=http://live.cap

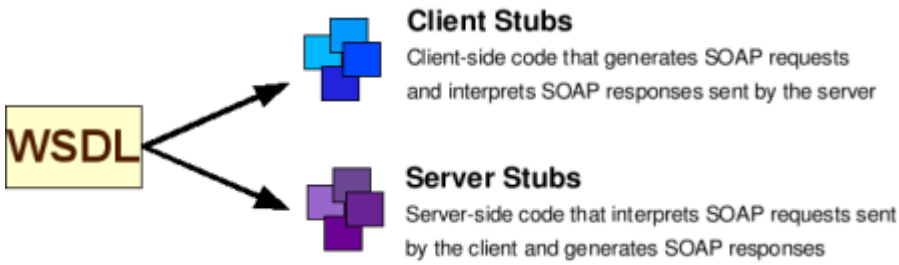
For example, if you visit the web interface (http://live.capescience.com/GlobalWeather) you'll see that the Weather Web service offers a `getWeatherReport` operation that expects a single string parameter (an IATA airport designation, e.g. ORD for Chicago O'Hare and LHR for London Heathrow). If you invoke `getWeatherReport`, the Web service will return a `WeatherReport` structure with all sorts of interesting weather data. Fun!

How does this work in practice?

OK, now that you have an idea of what Web Services are, you are probably anxious to start programming Web Services right away. Before you do that, you might want to know how Web Services-based applications are structured. If you've ever used CORBA or RMI, this structure will look pretty familiar.

First of all, you should know that despite having a lot of protocols and languages floating around, Web Services programmers usually only have to concentrate on writing code in their favorite programming language and, in some cases, in writing WSDL. SOAP code, on the other hand, is always generated and interpreted automatically for us. Once we've reached a point where our client application needs to invoke a Web Service, we *delegate* that task on a piece of software called a *stub*. The good news is that there are plenty of tools available that will generate stubs automatically for us, usually based on the WSDL description of the Web Service.

Figure 1-7. Client and server stubs are generated from the WSDL file



Using stubs simplifies our applications considerably. We don't have to write a complex client program that dynamically generates SOAP requests and interprets SOAP responses (and similarly for the server side of our application). We can simply concentrate on writing the client and/or server code, and leave all the dirty work to the stubs (which, again, we don't even have to write ourselves... they can be generated automatically from the WSDL description of a web service).

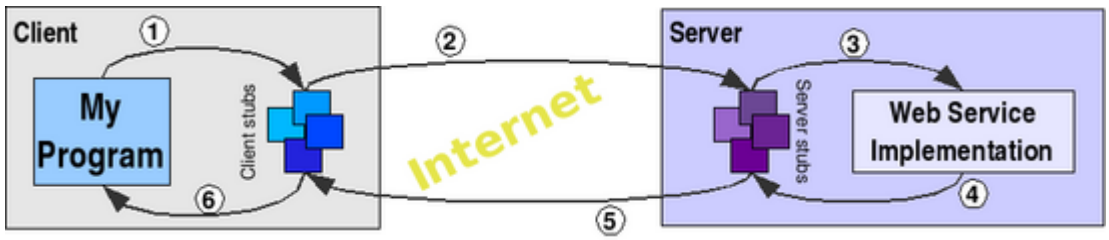
The stubs are generally generated only once. In other words, you shouldn't interpret the "Typical Web Service Invocation" figure (above) as saying that we go through the discovery process every single time we want to invoke a Web service, and generate the client stubs every time we want to invoke the service. In general, we only go through the discovery step once, then generate the stubs once (based on the WSDL of the service we've discovered) and then reuse the stubs as many times as we want (unless the

maintainers of the Web service decide to change the service's interface and, thus, its WSDL description). Of course, there are more complex invocation scenarios, but for now the one we've described is more than enough to understand how Web services work.

A Typical Web Service Invocation (redux)

So, let's suppose that we've already located the Web Service we want to use (either because we consulted a discovery service, or because the Web service URI was given to us), and we've generated the client stubs from the WSDL description. What exactly happens when we want to invoke a Web service operation from a program?

Figure 1-8. A typical Web Service invocation (more detailed)

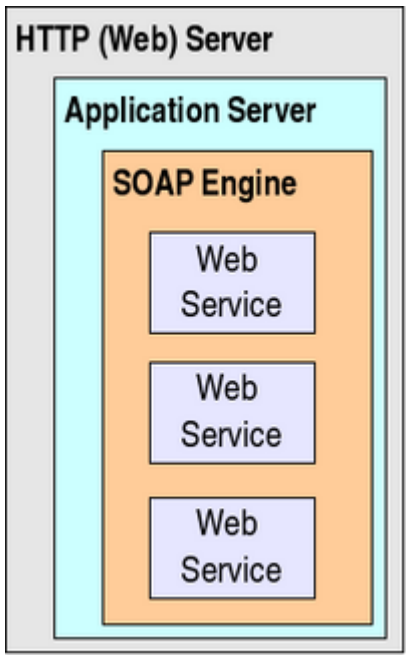


1. Whenever the client application needs to invoke the Web Service, it will really call the client stub. The client stub will turn this 'local invocation' into a proper SOAP request. This is often called the *marshaling* or *serializing* process.
2. The SOAP request is sent over a network using the HTTP protocol. The server receives the SOAP requests and hands it to the server stub. The server stub will convert the SOAP request into something the service implementation can understand (this is usually called *unmarshaling* or *deserializing*)
3. Once the SOAP request has been deserialized, the server stub invokes the service implementation, which then carries out the work it has been asked to do.
4. The result of the requested operation is handed to the server stub, which will turn it into a SOAP response.
5. The SOAP response is sent over a network using the HTTP protocol. The client stub receives the SOAP response and turns it into something the client application can understand.
6. Finally the application receives the result of the Web Service invocation and uses it.

The server side, up close

Finally, let's take a close look at what the server looks like, specially what software we should expect to have to get Web services up and running on our server.

Figure 1-9. The server side in a Web Services application



- **Web service**: First and foremost, we have our Web service. As we have seen, this is basically a piece of software that exposes a set of operations. For example, if we are implementing our Web service in Java, our service will be a Java class (and the operations will be implemented as Java methods). Obviously, we want a set of clients to be able to invoke those operations. However, our Web service implementation knows nothing about how to interpret SOAP requests and how to create SOAP responses. That's why we need a...
- **SOAP engine**: This is a piece of software that knows how to handle SOAP requests and responses. In practice, it is more common to use a generic SOAP engine than to actually generate server stubs for each individual Web service (note, however, that we still need client stubs for the client). One good example of a SOAP engine is Apache Axis (http://ws.apache.org/axis/) (this is, in fact, the SOAP engine used by the Globus Toolkit). However, the functionality of the SOAP engine is usually limited to manipulating SOAP. To actually function as a server that can receive requests from different clients, the SOAP engine usually runs within an...
- **Application server**: This is a piece of software that provides a 'living space' for applications that must be accessed by different clients. The SOAP engine runs as an application inside the application server. A good example is the Jakarta Tomcat (http://jakarta.apache.org/tomcat/) server, a Java Servlet and Java ServerPages container that is frequently used with Apache Axis and the Globus Toolkit.

  Many application servers already include some HTTP functionality, so we can have Web services up and running by installing a SOAP engine and an application server. However, when an application server lacks HTTP functionality, we also need an...
- **HTTP Server**: This is more commonly called a 'Web server'. It is a piece of software that knows how to handle HTTP messages. A good example is the Apache HTTP Server (http://httpd.apache.org/), one

of the most popular web servers in the Internet.

Note: Terminology in this area is still a bit inconsistent, so you might encounter different terms for the concepts we've just seen. In particular, it's very common to use the term *Web services container* as a catch-all term for the SOAP engine + application server + HTTP server.

WSRF: The Web Services Resource Framework

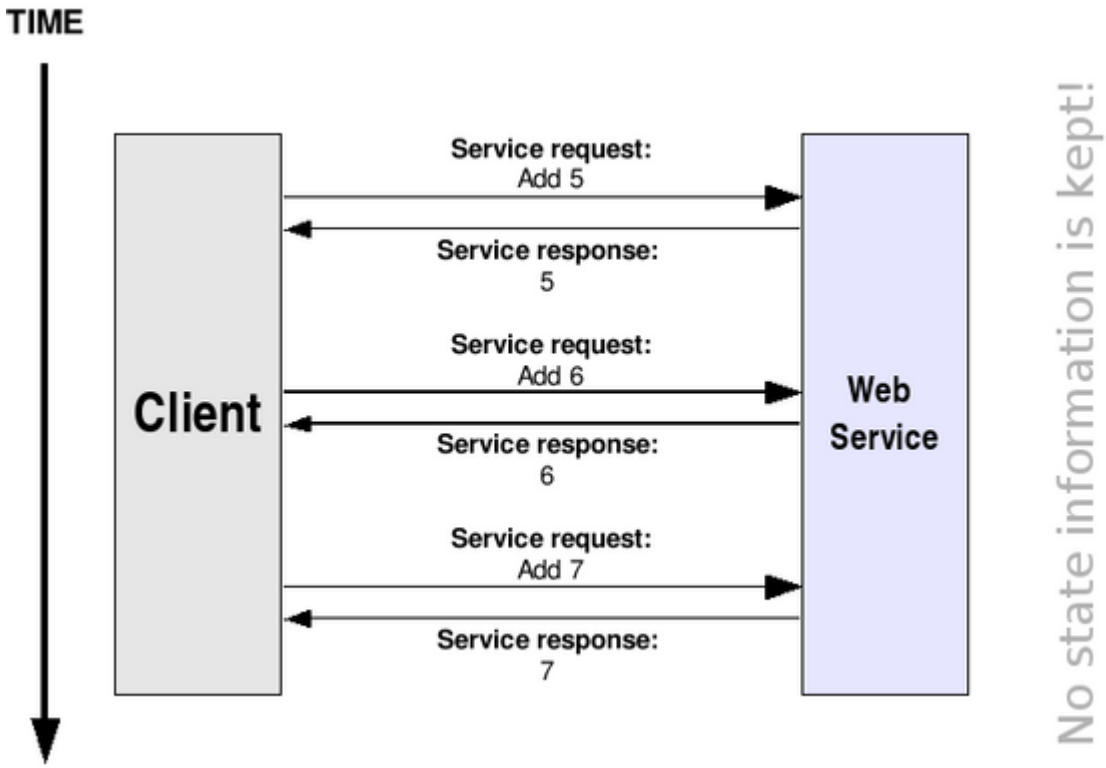
As we have just seen, Web Services are the technology of choice for Internet-based applications with loosely coupled clients and servers. That makes them the natural choice for building the next generation of grid-based applications. However, remember Web Services do have certain limitations. In fact, plain Web Services (as currently specified by the W3C) wouldn't be very helpful for building a grid application. Enter **WSRF**, which improves several aspects of web services to make them more adequate for grid applications.

In this section we'll take a brief look at the different parts of the WSRF specification. However, before doing that, we need to take a close look at the main improvement in WSRF: *statefulness*.

WSRF: It's all about state

Plain Web services are usually *stateless* (even though, in theory, there is nothing in the Web Services Architecture that says they can't be stateful). This means that the Web service can't "remember" information, or *keep state*, from one invocation to another. For example, imagine we want to program a very simple Web service which simply acts as an integer accumulator. This accumulator is initialized to zero, and we want to be able to add (accumulate) values in it. Suppose we have an `add` operation which receives the value to add and returns the current value of the accumulator. As shown in the following figure, our first invocation of this operation might seem to work (we request that 5 be added, and we receive 5 in return). However, since a Web service is stateless, the following invocations have no idea of what was done in the previous invocations. So, in the second call to `add` we get back 6, instead of 11 (which would be the expected value if the Web service was able to keep state).

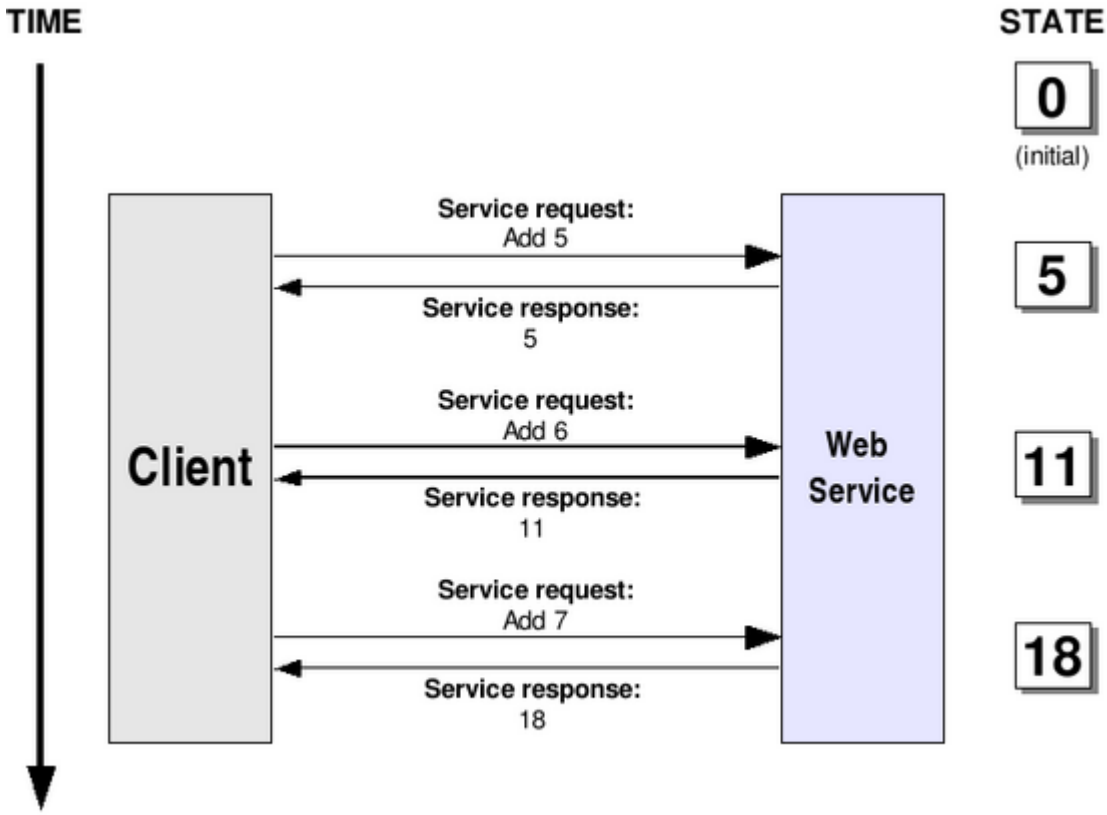
Figure 1-10. A stateless Web Service invocation



The fact that Web services don't keep state information is not necessarily a bad thing. There are plenty of applications which have no need whatsoever for statefulness. For example, the Weather Web service we saw in the previous section is a real, working Web service which has no need to know what happened in the previous invocations.

However, Grid applications *do* generally require statefulness. So, we would ideally like our Web service to somehow keep state information:

Figure 1-11. A stateful Web Service invocation



However, this is a pretty peculiar dilemma since, as mentioned above, a Web service is usually a stateless entity. In fact, some people might argue that a "stateful Web service" is a bit of a contradiction in terms! So, how do we get out of this jam?

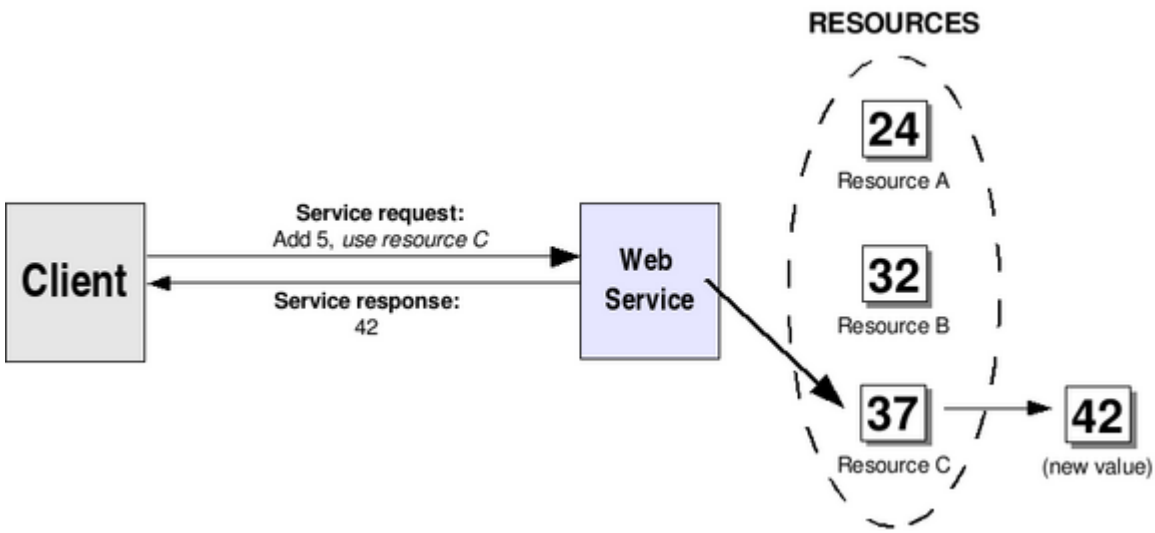
The resource approach to statefulness

Giving Web services the ability to keep state information while still keeping them stateless seems like a complex problem. Fortunately, it’s a problem with a very simple solution: simply keep the Web service and the state information completely separate.

Instead of putting the state *in* the Web service (thus making it stateful, which is generally regarded as a bad thing) we will keep it in a separate entity called a *resource*, which will store all the state information. Each resource will have a unique *key*, so whenever we want a *stateful interaction* with a Web service we simply have to instruct the Web service to use a particular resource.

For example, take the accumulator example. As shown in the next figure, our Web service could have three different resources (A, B, C) to choose from. If we want the integer value to be ’remembered’ from invocation to invocation, the client simply has to specify that he wants a method invoked *with* a certain resource.

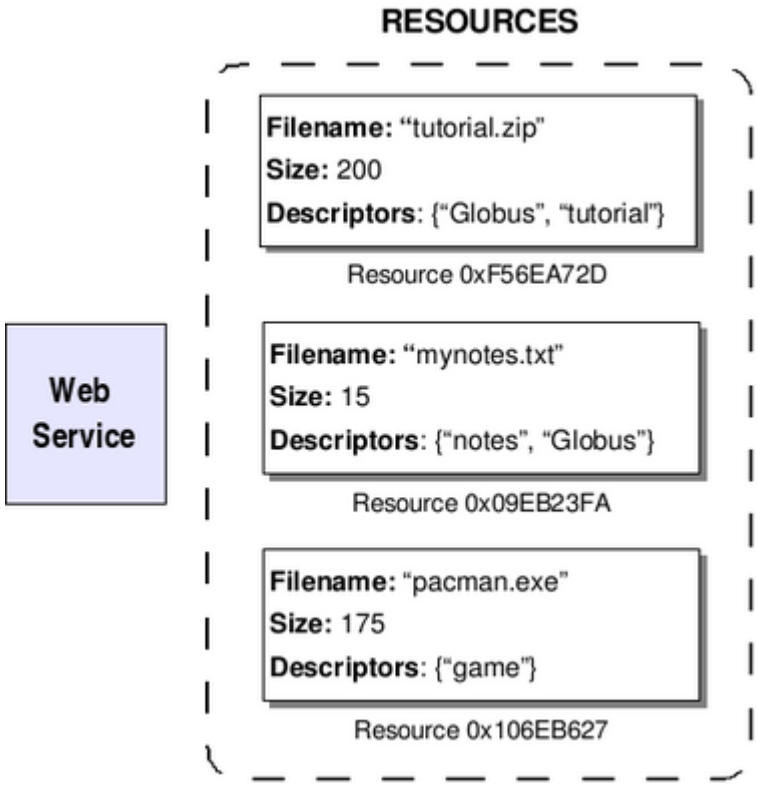
Figure 1-12. The resource approach to statefulness



In the figure we can see that the client wants the `add` operation invoked with resource C. When the Web service receives the `add` request, it will make sure to retrieve resource C so that `add` is actually performed on that resource. The resource themselves can be stored in memory, on secondary storage, or even in a database. Also, notice how a Web service can have access to more than one resource.

Of course, resources can come in all different shapes and sizes. A resource can keep multiple values (not just a simple integer value, as shown in the previous figure). For example, our resources could represent files:

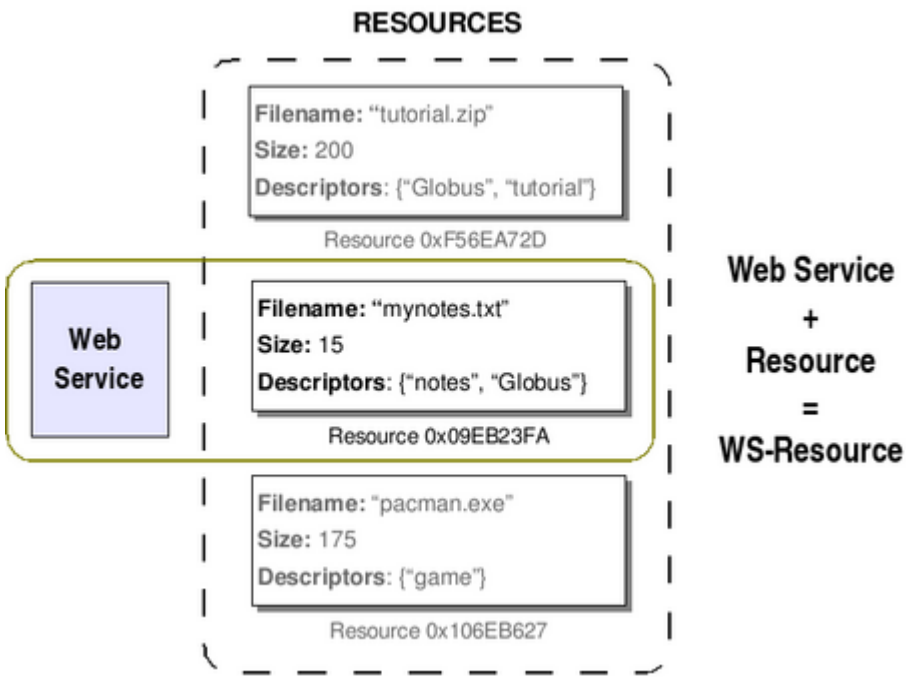
Figure 1-13. A Web Service with several resources. Each resource represents a file.



You might be wondering: And how exactly does the client specify what resource must be used? A URI might be enough to address the Web service, but how do we specify the resource on top of that? There are actually several different ways of doing this. As we'll see later on, the preferred way of doing it is to use a relatively new specification called WS-Addressing which provides a more versatile way of addressing Web Services (when compared to plain URIs).

Finally, a bit of terminology before we continue. A pairing of a Web service with a resource is called a WS-Resource. The address of a particular WS-Resource is called an *endpoint reference* (this is WS-Addressing lingo).

Figure 1-14. WS-Resource



The WSRF specification

The Web Services Resources Framework is a collection of five different specifications. Of course, they all relate (in some way or another) to the management of WS-Resources.

WS-ResourceProperties

A resource is composed of zero or more *resource properties*. For example, in the figure shown above each resource has three resource properties: Filename, Size, and Descriptors. WS-ResourceProperties specifies how resource properties are defined and accessed. As we'll see later on when we start programming, the resource properties are defined in the Web service's WSDL interface description.

WS-ResourceLifetime

Resources have non-trivial lifecycles. In other words, they're not a static entity that is created when our server starts and destroyed when our server stops. Resources can be created and destroyed at any time. The WS-ResourceLifetime supplies some basic mechanisms to manage the lifecycle of our resources.

WS-ServiceGroup

We will often be interested in managing *groups of Web Services* or *groups of WS-Resources*, and performing operations such as 'add new service to group', 'remove this service from group', and (more

importantly) 'find a service in the group that meets condition FOOBAR'. The WS-ServiceGroup specifies how exactly we should go about grouping services or WS-Resources together. Although the functionality provided by this specification is very basic, it is nonetheless the base of more powerful discovery services (such as GT4's IndexService) which allow us to group different services together and access them through a single point of entry (the service group).

WS-BaseFaults

Finally, this specification aims to provide a standard way of reporting faults when something goes wrong during a WS-Service invocation.

Related specifications

WS-Notification

WS-Notification is another collection of specifications that, although not a part of WSRF, is closely related to it. This specification allows a Web service to be configured as a *notification producer*, and certain clients to be *notification consumers* (or subscribers). This means that if a change occurs in the Web service (or, more specifically, in one of the WS-Resources), that change is *notified* to all the subscribers (not *all* changes are notified, only the ones the Web services programmer wants to).

WS-Addressing

As mentioned before, the WS-Addressing specification provides us a mechanism to address Web services which is much more versatile than plain URIs. In particular, we can use WS-Addressing to address a Web service + resource pair (a WS-Resource).

The Globus Toolkit 4

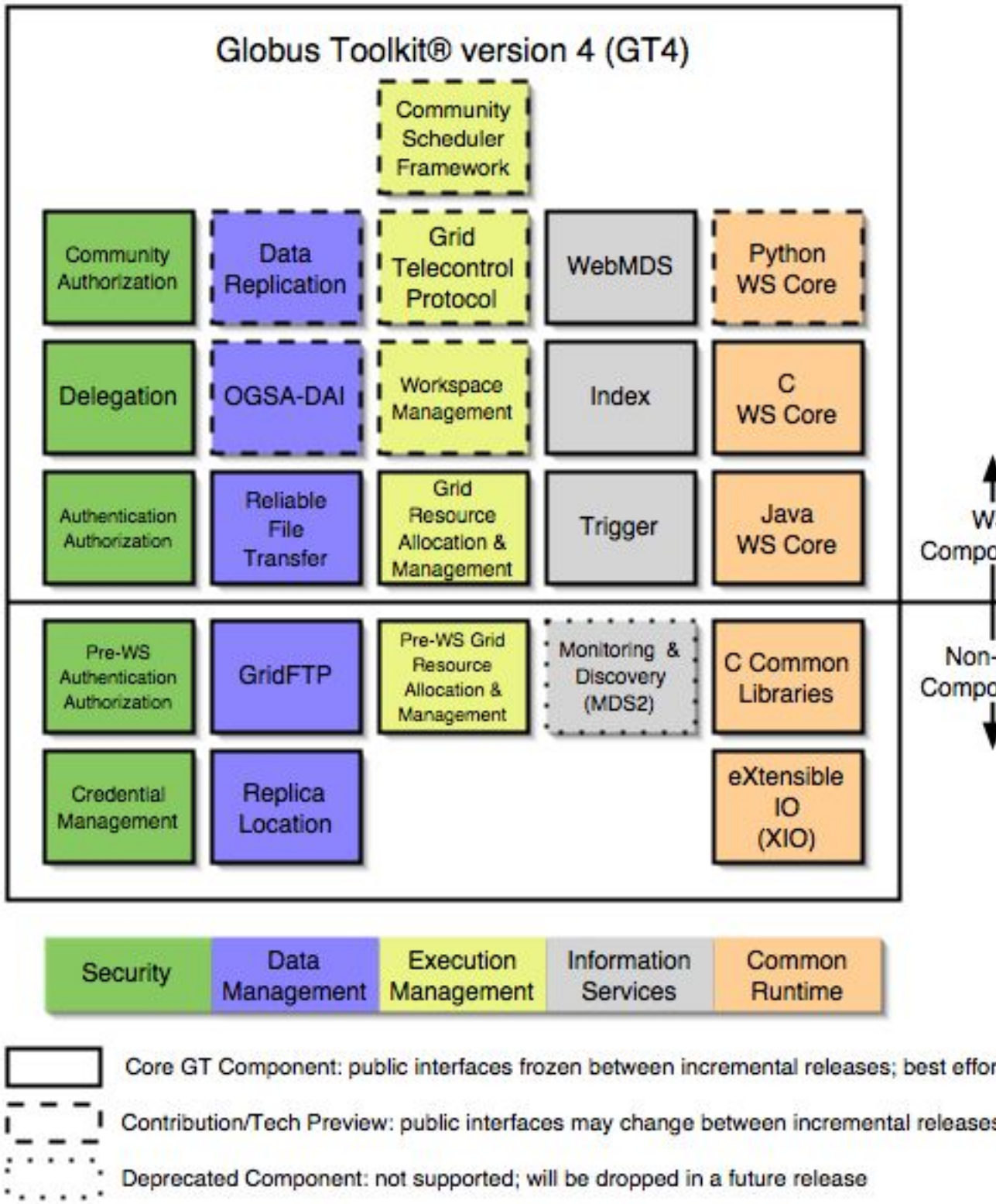
So, WSRF sure seems pretty cool and exciting, huh? However, if you've already programmed Grid-based applications, you're probably thinking that this is all very nice, but hardly enough for The Grid. Remember that WSRF is only a small (but important!) part of the whole GT4 Architecture: it is the *infrastructure* on top of which most of the toolkit is built. Besides the WSRF implementation, the toolkit includes a lot of components which we can use to program Grid applications.

Architecture

The Globus Toolkit 4 is composed of several software components. As shown in the following figure, these components are divided into five categories: Security, Data Management, Execution Management, Information Services, and the Common Runtime. Notice how, despite the fact that GT4 focuses on Web services, the toolkit also includes components which are *not* implemented on top of Web services. For

example, the GridFTP component uses a non-WS protocol which started as an ad hoc Globus protocol, but later became a GGF specification.

Figure 1-15. GT4 architecture



As mentioned in the preface, the tutorial currently focuses only on the Java WS Core component. Once again, it is important to realize that the Globus Toolkit includes a lot of other components which can help us build Grid systems. Even so, the Java WS Core component is specially interesting because it is the base for most of the WS components. Note that we do not need to have in-depth knowledge about Java WS Core to *use* many GT4 components like GRAM, MDS, etc. However, if we want to build a Grid system that *integrates* all of these components with our own services, we will need to know about Java WS Core to actually "glue" all those services together and to program our own services.

GT4 Components

Let’s take a quick look at what we can find in each of the five families of GT4 component. For more detailed descriptions, please refer to the official Globus documentation or to the Globus Primer (you can find a link in the preface)

Common Runtime

The Common Runtime components provide a set of fundamental libraries and tools which are needed to build both WS and non-WS services.

Security

Using the Security components, based on the Grid Security Infrastructure (GSI), we can make sure that our communications are secure.

Data management

These components will allow us to manage large sets of data in our virtual organization.

Information services

The Information Services, more commonly referred to as the Monitoring and Discovery Services (MDS), includes a set of components to discover and monitor resources in a virtual organization. Note that GT4 also includes a non-WS version of MDS (MDS2) for legacy purposes. This component is deprecated and will surely disappear in future releases of the toolkit.

Execution management

Execution Management components deal with the initiation, monitoring, management, scheduling and coordination of executable programs, usually called *jobs*, in a Grid.

Where to learn Java & XML

After seeing all the theory behind GT4, we're almost ready to start programming. However, remember you need to know Java to follow this tutorial. If you're new to Java, you will probably find the following sites interesting:

- The Java Tutorial (http://java.sun.com/docs/books/tutorial/) : The official tutorial from Sun, the makers of Java. Very good if you know absolutely nothing about Java.
- The Coffee Break (http://www.javacoffeebreak.com/) : Website with resources for Java programmers, including tutorials and FAQs.

Also, you need to be familiar with XML. You don't have to be an XML wizard, but should at least be able to read and interpret the different elements of an XML document. If you've never worked with XML, you should probably take a look at the following sites:

- W3Schools XML Tutorial (http://www.w3schools.com/xml/) : Tutorial that covers both the basics and the more advanced aspects of XML.
- ZVON.org (http://www.zvon.org/) : Tons of XML resources. Includes some very good reference guides.

Chapter 2. Installation

The tutorial doesn't include its own installation guide, since the official Globus installation guide is already very easy to follow and understand. Just follow the instructions in the GT4.0 System Administrator's Guide (http://www.globus.org/toolkit/docs/4.0/admin/docbook/). For the tutorial, you should be fine just installing the Java WS Core release of the toolkit. However, in the long run, you might prefer to have the complete toolkit install (be forewarned: the complete toolkit takes a *long* time to install).

II. GT4 Java WS Core

Chapter 3. Writing Your First Stateful Web Service in 5 Simple Steps

MathService

In this chapter we are going to write and deploy a simple stateful web service that uses WSRF to keep state information. Our first web service is an extremely simple *Math Web Service*, which we'll refer to as *MathService*. It will allow users to perform the following operations:

- Addition
- Subtraction

Furthermore, MathService will have the following resource properties (RPs for short):

- Value (integer)
- Last operation performed (string)

We will also add a "Get Value" operation to access the Value RP. In we will see a better way of accessing resource properties, without having to add get/set operations.

MathService's internal logic is very simple. Once a new resource is created, the "value" RP is initialized to zero, and the "last operation" RP is initialized to "NONE". The addition and subtraction operations expect only *one integer parameter*. This parameter is added/subtracted to the "value" RP, and the "last operation" RP is changed to "ADDITION" or "SUBTRACTION" accordingly. Also, the addition and subtraction operations don't return anything.

Finally, this first example will be limited to having *one single resource*. In the following chapters we will see how we can write a service that has several resources associated to it, as seen in .

High-tech stuff, huh? Don't worry if this seems a bit lackluster. Since this is going to be our first stateful web service, it's better to start with a small didactic service which we'll gradually improve by adding more complex resource properties, notifications, etc. You should always bear in mind that MathService is, after all, just a means to get acquainted with GT4. Typical WSRF Web Services are generally much more complex and do more than expose trivial operations (such as addition and subtraction).

The Five Steps

Writing and deploying a WSRF Web Service is easier than you might think. You just have to follow five simple steps.

1. **Define the service's interface**. This is done with *WSDL*

2. **Implement the service**. This is done with *Java*.
3. **Define the deployment parameters**. This is done with *WSDD* and *JNDI*
4. **Compile everything and generate a GAR file**. This is done with *Ant*
5. **Deploy service**. This is also done with *a GT4 tool*

Don't worry if you don't understand these five steps or are baffled by terms such as WSDL, WSDD, and Ant. In this first example we're going to go through each step in great detail, explaining what each step accomplishes, and giving detailed instructions on how to perform each step. The rest of the examples in the tutorial will also follow these five steps, but won't repeat the whole explanation of what that step is. So, if you ever find that you don't understand a particular step, you can always come back to this chapter ("Writing Your First Stateful Web Service in 5 Simple Steps") to review the details of that step.

Before we start...

Ready to start? Ok! Just hold your horses for a second. Don't forget to download the tutorial files before you start. You can find a link to the tutorial files in the tutorial website (http://gdp.globus.org/gt4-tutorial). The examples bundle includes all the tutorial source files, plus a couple of extra files we'll need to successfully build and deploy our service. Just create an empty directory on your file system and untar-gunzip the file there. From now on, we'll refer to that directory as **$EXAMPLES_DIR**.

Once you have the files, take into account that there are two ways of following the first chapters of the tutorial:

- **With the example source files**: You'll have all the source code (Java, WSDL, and WSDD) ready to use in `$EXAMPLES_DIR`, so there's no need to manually modify these files.
- **Without the examples source files**: Some people don't like getting all the source code ready to use out-of-the-box, but prefer to write the files themselves so they can have a better understanding of what they're doing at each point. In fact, we think this is probably the best way to follow this chapter (except for a few files which would take too long to write manually). Since this chapter includes complete code listings (which you can copy and paste to a file), you can easily write all the files yourself. However, you *do* need a set of auxiliary files included in the examples bundle that are needed to build and deploy the services. So, if you want to follow the examples without the source files, you still need to download the examples files. Once you're in `$EXAMPLES_DIR`, simply delete directory "`org`" to delete the source files, but *don't delete anything else*.

Ok, *now* we're ready to start :-)

Step 1: Defining the interface in WSDL

The first step in writing a web service (including those that use WSRF to keep state) is to define the *service interface*. We need to specify what our service is going to provide to the outer world. At this point we're not concerned with the inner workings of that service (what algorithms it uses, other systems

it interacts with, etc.). We just need to know what *operations* will be available to our users. In Web Services lingo, the service interface is usually called the *port type* (usually written *portType*).

As we saw in , there is a special XML language which can be used to specify what operations a web service offers: the Web Service Description Language (WSDL). So, what we need to do in this step is write a description of our MathService using WSDL.

At first sight, it might seem that starting with an interface language (such as a Java interface or an IDL interface) might be the best option, since (as you'll soon find out) it is more user-friendly than directly coding in WSDL. In fact, if we wanted to define our interface in Java, we could simply write the following:

```
public interface Math
{
  public void add(int a);

  public void subtract(int a);

  public int getValueRP();
}
```

...and we'd be nearly finished with step 1! (we would still need to specify the resource properties). However, we are going to start with a WSDL description of the interface, even if it is a bit harder to understand than using a Java interface. The main reason for this is that, although Java interfaces might be easier to write and understand, in the long run they produce much more problems than WSDL. So, the sooner we start writing WSDL, the better. Before that, we'll take a good look at the WSDL code which is equivalent to the Java interface shown above.

However, the goal of this page is not to give a detailed explanation of how to write a WSDL file, but rather to present the WSDL file for this particular example. If you have no idea whatsoever of how to write WSDL, now is a good time to take a look at . Come on, go take a look at the appendix. We'll be waiting for you right here.

The WSDL code

Ok, so supposing you either know WSDL or have visited the WSDL appendix, take a good thorough look at this WSDL code:

```
<?xml version="1.0" encoding="UTF-8"?>
<definitions name="MathService"
    targetNamespace="http://www.globus.org/namespaces/examples/core/MathService_instance"
    xmlns="http://schemas.xmlsoap.org/wsdl/"
    xmlns:tns="http://www.globus.org/namespaces/examples/core/MathService_instance"
    xmlns:wsdl="http://schemas.xmlsoap.org/wsdl/"
    xmlns:wsrp="http://docs.oasis-open.org/wsrf/2004/06/wsrf-WS-ResourceProperties-1.2-draf
    xmlns:wsrpw="http://docs.oasis-open.org/wsrf/2004/06/wsrf-WS-ResourceProperties-1.2-dra
    xmlns:wsdlpp="http://www.globus.org/namespaces/2004/10/WSDLPreprocessor"
    xmlns:xsd="http://www.w3.org/2001/XMLSchema">

<wsdl:import
    namespace=
```

```
    "http://docs.oasis-open.org/wsrf/2004/06/wsrf-WS-ResourceProperties-1.2-draft-01.wsdl"
    location="../../wsrf/properties/WS-ResourceProperties.wsdl" />


<!--============================================================

                    T Y P E S

  ============================================================-->
<types>
<xsd:schema targetNamespace="http://www.globus.org/namespaces/examples/core/MathService_ins
    xmlns:tns="http://www.globus.org/namespaces/examples/core/MathService_instance"
    xmlns:xsd="http://www.w3.org/2001/XMLSchema">


<!-- REQUESTS AND RESPONSES -->

<xsd:element name="add" type="xsd:int"/>
<xsd:element name="addResponse">
<xsd:complexType/>
</xsd:element>

<xsd:element name="subtract" type="xsd:int"/>
<xsd:element name="subtractResponse">
<xsd:complexType/>
</xsd:element>

<xsd:element name="getValueRP">
<xsd:complexType/>
</xsd:element>
<xsd:element name="getValueRPResponse" type="xsd:int"/>


<!-- RESOURCE PROPERTIES -->

<xsd:element name="Value" type="xsd:int"/>
<xsd:element name="LastOp" type="xsd:string"/>

<xsd:element name="MathResourceProperties">
<xsd:complexType>
<xsd:sequence>
<xsd:element ref="tns:Value" minOccurs="1" maxOccurs="1"/>
<xsd:element ref="tns:LastOp" minOccurs="1" maxOccurs="1"/>
</xsd:sequence>
</xsd:complexType>
</xsd:element>

</xsd:schema>
</types>


<!--============================================================
```

```
                        M E S S A G E S

  ============================================================-->
<message name="AddInputMessage">
<part name="parameters" element="tns:add"/>
</message>
<message name="AddOutputMessage">
<part name="parameters" element="tns:addResponse"/>
</message>

<message name="SubtractInputMessage">
<part name="parameters" element="tns:subtract"/>
</message>
<message name="SubtractOutputMessage">
<part name="parameters" element="tns:subtractResponse"/>
</message>

<message name="GetValueRPInputMessage">
<part name="parameters" element="tns:getValueRP"/>
</message>
<message name="GetValueRPOutputMessage">
<part name="parameters" element="tns:getValueRPResponse"/>
</message>



<!--============================================================

                        P O R T T Y P E

  ============================================================-->
<portType name="MathPortType"
    wsdlpp:extends="wsrpw:GetResourceProperty"
    wsrp:ResourceProperties="tns:MathResourceProperties">

<operation name="add">
<input message="tns:AddInputMessage"/>
<output message="tns:AddOutputMessage"/>
</operation>

<operation name="subtract">
<input message="tns:SubtractInputMessage"/>
<output message="tns:SubtractOutputMessage"/>
</operation>

<operation name="getValueRP">
<input message="tns:GetValueRPInputMessage"/>
<output message="tns:GetValueRPOutputMessage"/>
</operation>

</portType>

</definitions>
```

> **Note:** This file is `$EXAMPLES_DIR/schema/examples/MathService_instance/Math.wsdl`. This file is located in that particular directory because of the tool we'll be using to build the service. You can find more details about the directory structure required by this tool in .

If you know WSDL, you'll recognize this as a pretty straightforward WSDL file which defines three operations: `add`, `subtract`, and `getValueRP` (along with all the necessary messages and types). However, this WSDL file does have some peculiarities specific to WSRF and Globus.

WSRF and Globus-specific features of WSDL

Our WSDL file has three features which are specific to WSRF or to the Globus implementation of WSRF we're using. The following is just a brief overview of these three features. You can find more WSDL details in .

- **Resource properties**: We use the `wsrp:ResourceProperties` attribute of the `portType` element to specify what our service's resource properties are. The resource properties must be declared in the `<types>` section of the WSDL file. Remember that the resource properties are where we'll keep all our state information.
- **The WSDL Preprocessor**: Thanks to the `wsdlpp:extends` attribute of the `portType` element we can include existing WSRF portTypes in our own portType without having to copy-and-paste from the official WSRF WSDL files. A WSDL Preprocessor will use the value of that attribute to generate correct WSDL which includes our own portType definitions plus any WSRF portType we might need in our service. This is a Globus-specific feature that is included to make life easier for programmers.

  In our case, notice how we're including the the `GetResourceProperty` portType from the WS-ResourceProperties WSDL file.
- **No bindings**: Bindings are an essential part of a normal WSDL file. However, we don't have to add them manually, since they are generated automatically by a GT4 tool that is called when we build the service.

The WSDL Preprocessor

The WSDL Preprocessor (the `wsdlpp:extends` attribute in the `portType` element) is provided in GT4 as a *convenience*. Conceptually, it is very important to understand that WSDL files using `wsdlpp:extends` will always be converted to standard WSDL before they are actually used. In other words, `wsdlpp:extends` doesn't affect GT4's interoperability with other WSRF implementations because the `extends` attribute is *always* "purged" from our WSDL file before the service is deployed. This process is usually called "flattening" because we take several WSDL files (our file plus any WSRF WSDL files we extend from) and then merge them into a single (flattened) file. GT4 will always publish the *flattened* version of our WSDL file.

Also, take into account that you are not *required* to use `wsdlpp:extends`. If you choose to, you can write the flattened version directly. However, this involves a fair amount of copy-pasting that can be very error-prone. The book examples all use `wsdlpp:extends`.

Bottom line: GT4 doesn't require that other Web Services implementations or other WSRF implementations use `wsdlpp:extends` because WSDL files are *always* exchanged in their flattened versions. It is a purely internal feature of the toolkit.

Namespace mappings

One of the nice things about WSDL is that it's *language-neutral*. In other words, there is no mention of the language in which the service is going to be implemented, or of the language in which the client is going to be implemented.

However, there will of course come a moment when we'll want to refer to this interface from a specific language (in our case, Java). We do this through a set of *stub classes* (stubs were described in) which are generated from the WSDL file using a GT4 tool. For that tool to successfully generate the stub classes, we need to tell it where (i.e. in what Java package) to place the stub classes. We do this with a *mappings file*, which maps WSDL namespaces to Java packages:

```
http\://www.globus.org/namespaces/examples/core/MathService_instance=
    org.globus.examples.stubs.MathService_instance
http\://www.globus.org/namespaces/examples/core/MathService_instance/bindings=
    org.globus.examples.stubs.MathService_instance.bindings
http\://www.globus.org/namespaces/examples/core/MathService_instance/service=
    org.globus.examples.stubs.MathService_instance.service
```

Note: Each mapping must go in one line (i.e. the above file should have *three* lines). Also, take into account that the backslash before the colon is intentional. This file is `$EXAMPLES_DIR/namespace2package.properties`.

The first namespace is the target namespace of the WSDL file. The other two namespaces are automatically generated when a GT4 tool 'completes' the WSDL file (including the necessary bindings). Throughout the tutorial, the stub classes for the examples will be placed in the following Java package:

`org.globus.examples.stubs`

Since we're defining a service called `MathService_instance`, we're specifically mapping the WSDL file to the following package:

`org.globus.examples.stubs.`*`MathService_instance`*

However, take into account that the stubs classes are *generated* from the WSDL file, so they won't exist until we compile the service (which is when the stub classes are generated). In other words, don't look for the `org.globus.examples.stubs` package in `$EXAMPLES_DIR`, because you won't find them there. If you are of a curious disposition, don't worry: as soon as we generate the stub classes, we'll take a (very brief) look at the directory where they are generated.

Step 2: Implementing the service in Java

After defining the service interface ("*what* the service does"), the next step is implementing that interface. The implementation is "*how* the service does what it says it does".

The QNames interface

The first bit of code we need is a very simple Java interface that will make our life a bit easier. When we have to refer to just about anything related to a service, we will need to do so using its *qualified name*, or QName for short. This is a name which includes a namespace and a *local name*. For example, the QName of the `Value` RP is:

`{http://www.globus.org/namespaces/examples/core/MathService_instance}Value`

> **Note:** This is a common string representation of a QName. The namespace is placed between curly braces, and the local name is placed right after the namespace.

A qualified name is represented in Java using the `QName` class. Since we'll be referring to the service's qualified names frequently, it is a good practice to put them all in a separate interface:

```
package org.globus.examples.services.core.first.impl;

import javax.xml.namespace.QName;

public interface MathQNames {
public static final String NS = "http://www.globus.org/namespaces/examples/core/MathService

public static final QName RP_VALUE = new QName(NS, "Value");

public static final QName RP_LASTOP = new QName(NS, "LastOp");

public static final QName RESOURCE_PROPERTIES = new QName(NS,
"MathResourceProperties");
```

```
}
```

> **Note:** This file is `$EXAMPLES_DIR/org/globus/examples/services/core/first/impl/MathQNames.java`.

We are not really required to write this interface, but in the long run it's much better to have all the QNames in a single spot, so we can avoid making mistakes when manually writing QNames in our other classes.

The service implementation

In this first simple example, our service implementation will consist of a single Java class with the code for both the service *and* the resource. We will see in the following chapters that it is more common (in fact, desirable) to split the implementation into at least two classes: one for the service and another one for the resource. You should only use the approach described in this first example when coding very simple services.

Writing the code for the service is actually very mechanical. The only non-trivial piece of code is the method that will be in charge of initializing our service's single resource.

The bare bones of our resource class will be the following:

```
package org.globus.examples.services.core.first.impl;

import java.rmi.RemoteException;

import org.globus.wsrf.Resource;
import org.globus.wsrf.ResourceProperties;
import org.globus.wsrf.ResourceProperty;
import org.globus.wsrf.ResourcePropertySet;
import org.globus.wsrf.impl.ReflectionResourceProperty;
import org.globus.wsrf.impl.SimpleResourcePropertySet;
import org.globus.examples.stubs.MathService_instance.AddResponse;
import org.globus.examples.stubs.MathService_instance.SubtractResponse;
import org.globus.examples.stubs.MathService_instance.GetValueRP;

public class MathService implements Resource❶, ResourceProperties❷ {


}
```

❶ Since our Java class will implement both the service and the resource, we need to implement the `Resource` interface. However, this interface doesn't require any methods. It is simply a way of tagging a class as being a resource.

❷ By implementing the `ResourceProperties` interface we are indicating that our class has a set of resource properties which we want to make available. This interface requires that we add the following to our class:

```
private ResourcePropertySet propSet;
```

```
   public ResourcePropertySet getResourcePropertySet() {
   return this.propSet;
   }
```

Now, remember that our resource has two resource properties: `Value` of type `xsd:int` and `LastOp` of type `xsd:string`. We need to add an attribute for each resource property along with a get/set method pair for each resource property:

```
package org.globus.examples.services.core.first.impl;

import java.rmi.RemoteException;

import org.globus.wsrf.Resource;
import org.globus.wsrf.ResourceProperties;
import org.globus.wsrf.ResourceProperty;
import org.globus.wsrf.ResourcePropertySet;
import org.globus.wsrf.impl.ReflectionResourceProperty;
import org.globus.wsrf.impl.SimpleResourcePropertySet;
import org.globus.examples.stubs.MathService_instance.AddResponse;
import org.globus.examples.stubs.MathService_instance.SubtractResponse;
import org.globus.examples.stubs.MathService_instance.GetValueRP;

public class MathService implements Resource, ResourceProperties {

/* Resource Property set */
private ResourcePropertySet propSet;

/* Resource properties */
private int value;
private String lastOp;

/* Get/Setters for the RPs */
public int getValue() {
return value;
}

public void setValue(int value) {
this.value = value;
}

public String getLastOp() {
return lastOp;
}

public void setLastOp(String lastOp) {
this.lastOp = lastOp;
}

/* Required by interface ResourceProperties */
```

```
public ResourcePropertySet getResourcePropertySet() {
return this.propSet;
}
}
```

Caution

It is important for the attributes to have *the same name* that was given to the resource properties in the WSDL file (but with the first letter in lowercase). Remember:

```
<xsd:element name="Value" type="xsd:int"/>
<xsd:element name="LastOp" type="xsd:string"/>
```

This translates to:

```
private int value;
private String lastOp;
```

As for the get/set methods, we again have to use the same name used in the WSDL file (keeping the first letter in uppercase):

```
public int getValue() {...}
public void setValue(int value) {...}
public String getLastOp() {...}
public void setLastOp(String lastOp) {...}
```

Next, we have to implement the constructor. Here we will initialize the resource properties.

```
/* Constructor. Initializes RPs */
public MathService() throws RemoteException {
❶
this.propSet = new SimpleResourcePropertySet(
MathQNames.RESOURCE_PROPERTIES);

/* Initialize the RP's */
try {
❷
ResourceProperty valueRP = new ReflectionResourceProperty(
MathQNames.RP_VALUE, "Value", this);
this.propSet.add(valueRP);
setValue(0);

ResourceProperty lastOpRP = new ReflectionResourceProperty(
MathQNames.RP_LASTOP, "LastOp", this);
this.propSet.add(lastOpRP);
setLastOp("NONE");
} catch (Exception e) {
throw new RuntimeException(e.getMessage());
}
}
```

❶ We create the resource property set. To do so, we need to provide the qualified name of the resource properties. In our case, that QName is:
`{http://www.globus.org/namespaces/examples/core/MathService_instance}MathResourcePropert`

This was specified in the WSDL file. Remember that we put this QName in the `MathQNames` interface so we could access it with ease.

❷ We create the individual resource properties, and initialize them (again, notice how we're referring to each resource property's QName: `RP_VALUE` and `RP_LASTOP`)

> **ReflectionResourceProperty:** What we have just seen is the simplest type of resource creation: using `ReflectionResourceProperty` to represent each resource property. This makes the implementation much simpler, but also adds quite a bit of restrictions on our resource implementation (such as the need for get/set methods, as outlined in the caution box above). GT4 includes other classes to deal with more complex resource scenarios (such as `SimpleResourceProperty`, `PersistentResourceProperty`, etc.). In fact, we will start using `SimpleResourceProperty` in .

Finally, we need to provide the implementation of our remotely-accessible methods (`add`, `subtract`, and `getValueRP`). These are pretty straightforward, except for some peculiarities in how the parameters and return types have to be declared. For example, take a look at the `add` method: (the `subtract` method is similar)

```
public AddResponse add(int a) throws RemoteException {
value += a;
lastOp = "ADDITION";

return new AddResponse();
}
```

You'll notice that, even though we defined the `add` operation as having no return type in the WSDL file, now the return type is `AddResponse`. A similar thing happens in the `getValueRP` method:

```
public int getValueRP(GetValueRP params) throws RemoteException {
return value;
}
```

Even though `getValueRP` was defined as having no parameters, it turns out our method has a single "`GetValueRP params`" parameter. Don't get nervous... this all has a very logical explanation. This is due to the fact that WSRF uses *document/literal bindings*, which requires that the parameters be implemented in a very particular way.

> **How document/literal bindings affect our parameters:** Whenever we write an operation which is part of our WSDL interface (such as `add`, `subtract`, or `getValueRP`), the parameters and the return values will *in some cases* be 'boxed' inside stub classes (which are generated automatically from the WSDL file). This is more evident when we have several parameters. For example, if we declared the following operation in our WSDL file:
>
> ```
> void multiply(int a1, int a2);
> ```

The actual Java code would look like this:

```
public MultiplyResponse multiply(Multiply params) throws RemoteException
{
int a1 = params.getA1()
int a2 = params.getA2()

// Do something

return new MultiplyResponse();
}
```

`Multiply` and `MultiplyResponse` are stub classes. Notice how the two parameters (`a1` and `a2`) are 'boxed' inside a single `Multiply` parameter, and how we return a `MultiplyResponse` object, even though we don't really want to return anything.

The tricky thing about this is that, as mentioned earlier, this 'boxing' process only happens in some cases:

- When the number of parameters is more than one. For example, our `add` method has a single parameter, so it is *not* 'boxed'.
- When the return type is void or a complex type. For example, our `getValueRP` method returns an `int` value, so it is not 'boxed'. On the other hand, both `add` and `subtract` return void, so what we really have to return is `AddResponse` and `SubtractResponse`.

The complete class would look like this:

```
package org.globus.examples.services.core.first.impl;

import java.rmi.RemoteException;

import org.globus.wsrf.Resource;
import org.globus.wsrf.ResourceProperties;
import org.globus.wsrf.ResourceProperty;
import org.globus.wsrf.ResourcePropertySet;
import org.globus.wsrf.impl.ReflectionResourceProperty;
import org.globus.wsrf.impl.SimpleResourcePropertySet;
import org.globus.examples.stubs.MathService_instance.AddResponse;
import org.globus.examples.stubs.MathService_instance.SubtractResponse;
import org.globus.examples.stubs.MathService_instance.GetValueRP;

public class MathService implements Resource, ResourceProperties {

/* Resource Property set */
private ResourcePropertySet propSet;

/* Resource properties */
private int value;
private String lastOp;

/* Constructor. Initializes RPs */
public MathService() throws RemoteException {
```

```
/* Create RP set */
this.propSet = new SimpleResourcePropertySet(
MathQNames.RESOURCE_PROPERTIES);

/* Initialize the RP's */
try {
ResourceProperty valueRP = new ReflectionResourceProperty(
MathQNames.RP_VALUE, "Value", this);
this.propSet.add(valueRP);
setValue(0);

ResourceProperty lastOpRP = new ReflectionResourceProperty(
MathQNames.RP_LASTOP, "LastOp", this);
this.propSet.add(lastOpRP);
setLastOp("NONE");
} catch (Exception e) {
throw new RuntimeException(e.getMessage());
}
}

/* Get/Setters for the RPs */
public int getValue() {
return value;
}

public void setValue(int value) {
this.value = value;
}

public String getLastOp() {
return lastOp;
}

public void setLastOp(String lastOp) {
this.lastOp = lastOp;
}

/* Remotely-accessible operations */

public AddResponse add(int a) throws RemoteException {
value += a;
lastOp = "ADDITION";

return new AddResponse();
}

public SubtractResponse subtract(int a) throws RemoteException {
value -= a;
lastOp = "SUBTRACTION";

return new SubtractResponse();
}
```

```
public int getValueRP(GetValueRP params) throws RemoteException {
return value;
}

/* Required by interface ResourceProperties */
public ResourcePropertySet getResourcePropertySet() {
return this.propSet;
}
}
```

> **Note:** This file is `$EXAMPLE_DIR/org/globus/examples/services/core/first/impl/MathService.java`.

Step 3: Configuring the deployment in WSDD (and JNDI)

Up to this point, we have written the two most important parts of our stateful Web service: the service interface (WSDL) and the service implementation (Java). However, we still seem to be missing something... How do we actually make our web service available to client connections? Does our Java class simply float around in some sort of mysterious ether? This next step will actually take all the loose pieces we have written up to this point and make them available through a *Web services container*. This step is called the *deployment* of the web service.

> **Note:** Remember: The "Web Services container" is a catch-all term referring to all the software (SOAP Engine, Application Server, and HTTP Server) we need to make Web services available to clients. This might be a good moment to review .

The WSDD deployment descriptor

One of the key components of the deployment phase is a file called the *deployment descriptor*. It's the file that tells the Web Services container how it should publish our web service (for example, telling it what the our service's URI will be). The deployment descriptor is written in WSDD format (Web Service Deployment Descriptor). The deployment descriptor for our Web service will look like this:

```
<?xml version="1.0" encoding="UTF-8"?>
<deployment name="defaultServerConfig"
    xmlns="http://xml.apache.org/axis/wsdd/"
    xmlns:java="http://xml.apache.org/axis/wsdd/providers/java"
    xmlns:xsd="http://www.w3.org/2001/XMLSchema">

    <service name="examples/core/first/MathService" provider="Handler" use="literal" style=
        <parameter name="className" value="org.globus.examples.services.core.first.impl.Mat
        <wsdlFile>share/schema/examples/MathService_instance/Math_service.wsdl</wsdlFile>
        <parameter name="allowedMethods" value="*"/>
```

```
        <parameter name="handlerClass" value="org.globus.axis.providers.RPCProvider"/>
        <parameter name="scope" value="Application"/>
        <parameter name="providers" value="GetRPProvider"/>
        <parameter name="loadOnStartup" value="true"/>
    </service>

</deployment>
```

> **Note:** This file is `$EXAMPLES_DIR/org/globus/examples/services/core/first/deploy-server.wsdd`.

Let's take a close look at what all this means...

The 'service name'

```
<service name="examples/core/first/MathService" provider="Handler" use="literal" style="doc
```

This specifies the location where our web service will be found. If we combine this with the base address of our Web Services container, we will get the full URI of our web service. For example, if we are using the GT4 standalone container, the base URL will probably be `http://localhost:8080/wsrf/services`. Therefore, our service's URI would be:

```
http://localhost:8080/wsrf/services/examples/core/first/MathService
```

className

```
<parameter name="className" value="org.globus.examples.services.core.first.impl.MathService
```

This parameter refers to the class which implements the service interface (in our case, `MathService` from the previous section).

The WSDL file

```
<wsdlFile>share/schema/examples/MathService_instance/Math_service.wsdl</wsdlFile>
```

The `wsdlFile` tag tells the Web Services container where the WSDL file for this service can be found. Notice how there's a "`_service`" at the end of the filename. This is not a typo. This WSDL file (`Math_service.wsdl`) will be generated automatically by a GT4 tool when we compile the service.

The operation providers

For now, you can safely ignore the `providers` parameter, as we will not be using it in this example.

Load on startup

```
<parameter name="loadOnStartup" value="true"/>
```

This parameter allows us to control if we want the service to be loaded as soon as the container is started. Since our service has a single resource, it is usually best to load it at startup.

The common parameters

```
<parameter name="allowedMethods" value="*"/>
<parameter name="handlerClass" value="org.globus.axis.providers.RPCProvider"/>
<parameter name="scope" value="Application"/>
```

These are three parameters which we'll see in every web service we program and are better left untouched.

The JNDI deployment file

This file barely comes into play in this example since we're implementing our service the simplest possible way. However, we still have to include this file, but we need you to take a little leap of faith at this point and just accept that we need the file "because we need it". In the next chapter we will introduce the concept of *resource homes* and we will explain this file in more detail (we will also revisit the file seen in this example).

So, the JNDI deployment file looks like this:

```
<?xml version="1.0" encoding="UTF-8"?>
<jndiConfig xmlns="http://wsrf.globus.org/jndi/config">

<service name="examples/core/first/MathService">
<resource name="home" type="org.globus.wsrf.impl.ServiceResourceHome">
<resourceParams>

<parameter>
<name>factory</name>
<value>org.globus.wsrf.jndi.BeanFactory</value>
</parameter>

</resourceParams>

</resource>
</service>

</jndiConfig>
```

> **Note:** This file is `$EXAMPLES_DIR/org/globus/examples/services/core/first/deploy-jndi-config.xml`.

Step 4: Create a GAR file with Ant

At this point we have (1) a service interface in WSDL, (2) a service implementation in Java, and (3) a deployment descriptor in WSDD and JNDI telling the Web Services container how to present (1) and (2) to the outer world. However, all this is a bunch of loose files. How are we supposed to place this in a Web services container? Do we have to copy these files to strategically located directories? And what about the Java files? We haven't compiled those yet!

Fear not, for this is the step when everything comes together in perfect harmony. Using those three files we wrote in the previous three pages we will generate a *Grid Archive*, or *GAR file*. This GAR file is a single file which contains all the files and information the Web services container needs to *deploy* our service and make it available to the whole world. In fact, in the next section we'll instruct the GT4 standalone container to take the GAR and deploy it.

However, creating a GAR file is a pretty complex task which involves the following:

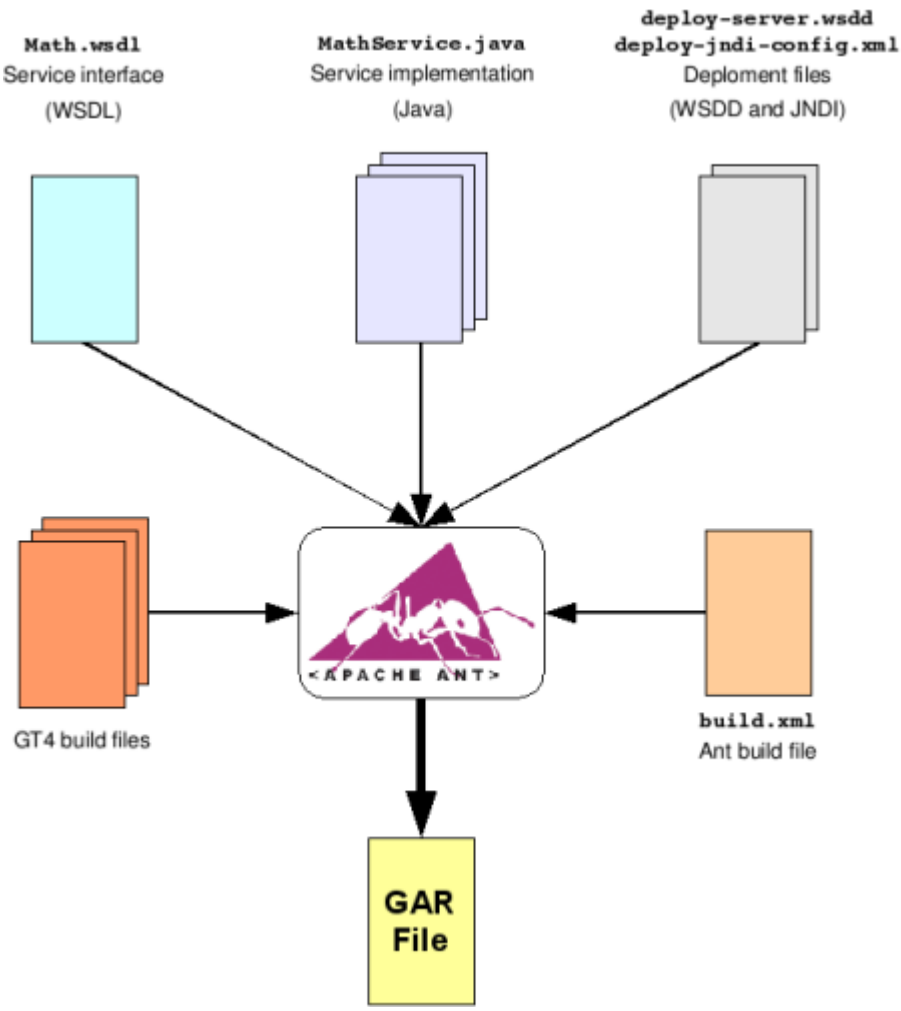
- Processing the WSDL file to add missing pieces (such as bindings)
- Creating the stub classes from the WSDL
- Compiling the stubs classes
- Compiling the service implementation
- Organize all the files into a very specific directory structure

Don't be scared by all this. Thanks to the hard work of the Globus guys and gals, we can do all this in a single step using a very useful tool called Ant.

Ant

Ant, an Apache Software Foundation (http://www.apache.org/) project, is a Java *build tool*. In concept, it is very similar to the classic UNIX `make` command. It allows programmers to forget about the individual steps involved in obtaining an executable from the source files, which will be taken care of by Ant. Each project is different, so the individual steps are described in a *buildfile* ('Makefile' in the make jargon). This buildfile directs Ant on what it should compile, how it should compile it, and in what order. This simplifies the whole process considerably. In fact, it reduces the number of steps to one! With Ant, all we have to worry about is writing the service interface, the service implementation, and the deployment descriptor. Ant takes care of the rest:

Figure 3-1. Generating a GAR file with Ant



As you can see, Ant generates the GAR directly from the three sets of source files. Internally, it is carrying out all the steps listed earlier, sparing us the cumbersome task of doing them ourselves. In a GT4 project, Ant uses two sets of buildfiles: a couple of buildfiles which are a part of GT4, and a buildfile we'll have to write on our own. The GT4 buildfiles cover all the important steps (generating the WSDL code, generating the stubs, ...). Our build file essentially has all the unique parameters of our web service, and a bunch of calls to the GT4 buildfiles.

Finally, if you want to learn more about Ant, take a look at the Ant Website (http://ant.apache.org/). It includes plenty of documentation, tutorials, etc.

The `globus-build-service` script and buildfile

Throughout the tutorial, we won't have to write a separate buildfile for each of our services. We will be relying on the `globus-build-service` script and buildfile, one of the tools developed as part of the Globus Service Build Tools (GSBT) project. This tool will allow us to create a GAR file with minimal

effort, and without having to modify an Ant buildfile every time we move on to the next example. A copy of `globus-build-service` is included with the examples bundle, and more information on the tool can be found on the GSBT website (http://gsbt.sourceforge.net/).

Creating the MathService GAR

Using the provided Ant buildfile and the handy script, building a web service is as simple as doing the following:

```
./globus-build-service.sh -d <service base directory> -s <service's WSDL file>
```

> **Note:** Make sure you have an environment variable called GLOBUS_LOCATION pointed to your Globus Toolkit root (the script depends on this)

> **Note:** Windows users can use a Python build script included with the downloadable tutorial files (and also a part of the GSBT project). If you prefer to use the Python script, simply replace `globus-build-service.sh` with `globus-build-service.py` in all the following examples.

The "service base directory" is the directory where we placed the `deploy-server.wsdd` file, and where the Java files can be found (inside an `impl` directory). To build the first example we simply need to do the following:

```
./globus-build-service.sh \
-d org/globus/examples/services/core/first/ \
-s schema/examples/MathService_instance/Math.wsdl
```

`globus-build-service` also allows us to use a shorthand notation which is much easier (and faster) to use. For example, to build our first example and generate its GAR file, we simply need to do the following:

```
./globus-build-service.sh first
```

> **Note:** Make sure you run this from `$EXAMPLES_DIR`.

We will be able to use this shorthand notation with all the examples included in the tutorial. However, this shorthand notation will work because the examples bundle includes a file that maps an abbreviated name (like `first`) to a specific directory and schema file. To write your own mappings and use the shorthand notation in your own projects, refer to the Globus Service Build Tools (http://gsbt.sourceforge.net/) website.

If everything works fine, the GAR file will be placed in `$EXAMPLES_DIR`. To be exact, the GAR file generated for this example will be the following:

```
$EXAMPLES_DIR/org_globus_examples_services_core_first.gar
```

Step 5: Deploy the service into a Web Services container

The GAR file, as mentioned in the previous page, contains all the files and information the web server needs to deploy the web service. Deployment is done with a GT4 tool that, using Ant, unpacks the GAR file and copies the files within (WSDL, compiled stubs, compiled implementation, WSDD) into key locations in the GT4 directory tree.

This deployment command must be run with a user that has write permission in $GLOBUS_LOCATION.

```
globus-deploy-gar $EXAMPLES_DIR/org_globus_examples_services_core_first.gar
```

There is also a command to *undeploy* a service:

```
globus-undeploy-gar org_globus_examples_services_core_first
```

Deployment is really as simple as that! That also concludes the five steps necessary to write and deploy a WSRF Web service. However, although you're probably beaming with pride because you've deployed your first WSRF web service, you'll certainly want to make sure that it works. We'll try out our recently deployed service using a very simple client application.

A simple client

We're going to test our web service with a command-line client which will invoke both the `add` and `subtract` operations and will also retrieve the `Value` resource property using the `getValueRP` operation. This client expects one argument from the command line:

1. The service URI

The client class will be called `Client` and we'll place it in the `$EXAMPLES_DIR/org/globus/examples/clients/MathService_instance/Client.java` file. The full code for the client is the following:

```
package org.globus.examples.clients.MathService_instance;

import org.apache.axis.message.addressing.Address;
import org.apache.axis.message.addressing.EndpointReferenceType;

import org.globus.examples.stubs.MathService_instance.MathPortType;
import org.globus.examples.stubs.MathService_instance.GetValueRP;
import org.globus.examples.stubs.MathService_instance.service.MathServiceAddressingLocator;

public class Client {

public static void main(String[] args) {
MathServiceAddressingLocator locator = new MathServiceAddressingLocator();

try {
```

```
String serviceURI=args[0];

❶
EndpointReferenceType endpoint = new EndpointReferenceType();
endpoint.setAddress(new Address(serviceURI));

❷
MathPortType math = locator.getMathPortTypePort(endpoint);

❸
// Perform an addition
math.add(10);

// Perform another addition
math.add(5);

// Access value
System.out.println("Current value:" + math.getValue(new GetValueRP()));

// Perform a subtraction
math.subtract(5);

// Access value
System.out.println("Current value:" + math.getValue(new GetValueRP()));
} catch (Exception e) { ❹
e.printStackTrace();
}
}

}
```

❶ First, we create an `EndpointReferenceType` object representing the endpoint reference of this service. Remember from that an endpoint reference is used to address a particular WS-Resource (the pairing of a service and a resource). Since our service has a single resource, our endpoint reference only needs the service's URI.

❷ Next, we obtain a reference to the service's portType. This is done with a stub class called `MathServiceAddressingLocator` that, given the service's endpoint, returns a `MathPortType` object that will allow us to contact the `Math` portType.

❸ Once we have that reference, we can work with the web service *as if it were a local object*. For example, to invoke the *remote* `add` operation, we simply have to use the `add` method in the `MathPortType` object.

❹ Finally, notice how all the code must be placed inside a try/catch block. We must always do this, since all the remote operations can throw `RemoteException`s (for example, if there is a network failure and we can't contact the service).

We are now going to compile the client. Before running the compiler, make sure you run the following:

```
source $GLOBUS_LOCATION/etc/globus-devel-env.sh
```

The `globus-devel-env.sh` script takes care of putting all the Globus libraries into your CLASSPATH. When compiling the service, Ant took care of this but, since we're not using Ant to compile the client, we need to run the script.

To compile the client, do the following:

```
javac \
-classpath ./build/stubs/classes/:$CLASSPATH \
org/globus/examples/clients/MathService_instance/Client.java
```

`./build/classes` is a directory generated by Ant were all the compiled stub classes are placed. We need to include this directory in the CLASSPATH so our client can access generated stub classes such as `MathServiceAddressingLocator`.

Now, before running the client, we need to to start up the standalone container. Otherwise, our web service won't be available, and the client will crash.

```
globus-start-container -nosec
```

> **Caution**
>
> We are running the Globus standalone container without any security (`-nosec`) to avoid having to deal with all the messy security configuration at this point. However, 'real' Grid application will almost *always* require security. The tutorial currently does not cover security (you will need to refer to the official Globus documentation).

When the container starts up, you'll see a list with the URIs of all the deployed services. One quick way of checking if MathService has been correctly deployed is to check if the following line appears in the list of services:

```
http://127.0.0.1:8080/wsrf/services/examples/core/first/MathService
```

> **Note:** This is the service as it would appear in a default GT4 installation, with the standalone container located in `http://localhost:8080/wsrf/services`. The URI might be different if you've changed the location of the container.

If the service is correctly deployed, we can now run the client:

```
java \
-classpath ./build/stubs/classes/:$CLASSPATH \
org.globus.examples.clients.MathService_instance.Client \
http://127.0.0.1:8080/wsrf/services/examples/core/first/MathService
```

If all goes well, you should see the following:

```
Current value: 15
Current value: 10
```

Now, remember that our service is, at the same time, the resource itself. So, if we invoke the service repeatedly, we will access the same stateful information. If you run the client a couple more times, you should see the value increase with each run:

```
Current value: 25
Current value: 20

Current value: 35
Current value: 30
```

Chapter 4. Singleton resources

In the previous chapter, we saw how to implement a single-resource stateful web service. We did this the simplest possible way: implementing the service *and* the resource in the same class. In this chapter, although we will continue to have a single resource, we will learn more about the preferred way of implementing web services in GT4: using a separate class for the service and the resource. To do this, we will learn more about *resource homes*.

Splitting up the implementation

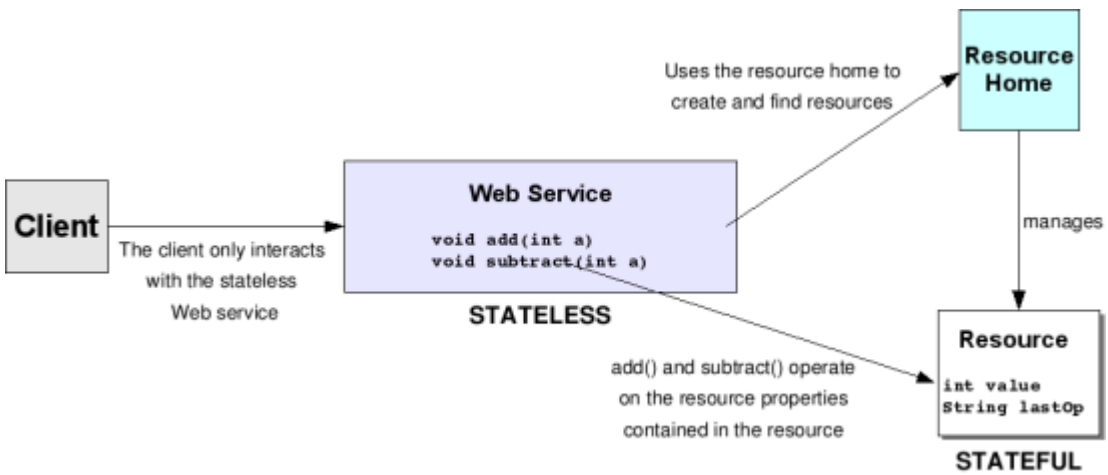
The resource, the home, and the service

In this chapter, we will split our implementation into three files. We will see that, for the most part, we will simply take code from the previous chapter and divide it among the three files.

- The resource:
  `$EXAMPLES_DIR/org/globus/examples/services/core/singleton/impl/MathResource.java`
- The resource home:
  `$EXAMPLES_DIR/org/globus/examples/services/core/singleton/impl/MathResourceHome.java`
- The service itself:
  `$EXAMPLES_DIR/org/globus/examples/services/core/singleton/impl/MathService.java`

Before looking at the actual Java code, let's make sure we understand how these three implementation files are related.

Figure 4-1. Relationships between the Service, the Resource Home, and the Resource

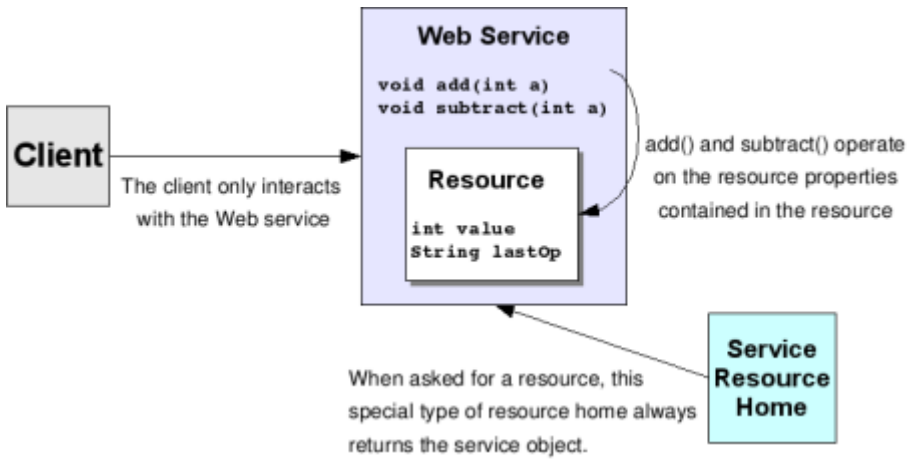


- The **service** is the *stateless* frontend (the client only interacts with this class, even if there are more classes lurking around in the background). Remember from section that, since we're following a 'resource approach' to statefulness, the web service will *always* be stateless. However, we can give the impression of being stateful by retrieving a *stateful resource* whenever we want to access state information.
- The **resource** is the *stateful* class were we keep all our information.
- The **resource home** is in charge of *managing* the resources. For example, the (stateless) service will use the resource home to retrieve the (stateful) resource. In this chapter, the resource home will be very simple since it will only deal with a single (or *singleton*) resource. In the next chapter, we will see how to set up a resource home that can manage several resources.

ServiceResourceHome: You might be wondering how this related to the example we saw in the previous chapter. Even though we did not implement a resource home, there *was* a resource home lurking around in the background. Remember the following line from the JNDI config file?

```
<resource name="home" type="org.globus.wsrf.impl.ServiceResourceHome">
```

We used a Globus-supplied resource home called `ServiceResourceHome`. As the following figure shows, the `ServiceResourceHome` is a special type of resource home that always returns the service object when asked for a resource. This allows us to implement our resource and service in the same class.



For simplicity, we will use `ServiceResourceHome` in other examples in the tutorial, as it will spare us a lot of code. However, remember that the preferred way of implementing services is by splitting up the implementation, as we will do in this chapter. When you start writing your own services, you should only use `ServiceResourceHome` for very simple services.

The WSDL file

In this chapter, we are only changing the *implementation* of our service. The interface is still the same, so there's no need to modify the WSDL file. We can reuse the one from the previous chapter.

Note: The WSDL file is `$EXAMPLES_DIR/schema/examples/MathService_instance/Math.wsdl`

The QNames interface

Once again, the first bit of code we are going to write is the namespaces interface. Again, since we are reusing the WSDL file from the previous chapter, there's no big changes to the `MathQNames` interface, except for the fact that we are now placing our Java classes in a new package.

```
package org.globus.examples.services.core.singleton.impl;

import javax.xml.namespace.QName;

public interface MathQNames {
public static final String NS = "http://www.globus.org/namespaces/examples/core/MathService

public static final QName RP_VALUE = new QName(NS, "Value");

public static final QName RP_LASTOP = new QName(NS, "LastOp");

public static final QName RESOURCE_PROPERTIES = new QName(NS,
"MathResourceProperties");
}
```

Note: This file is `$EXAMPLES_DIR/org/globus/examples/services/core/singleton/impl/MathQNames.java`.

In the following chapters, we won't see the complete code of each new `MathQNames` interface. As you see, it is pretty straightforward to write it from the WSDL file. From now on, we will simply point you to the location of the file in the examples bundle.

The resource resource

The first big class we will see in the resource implementation. You will be (pleasantly) surprised to see that it is practically identical to the `MathService` class from the previous chapter. The only thing missing is the `add`, `subtract`, and `getValueRP` methods, which we will now place in the service implementation.

Also, notice how all the RP initialization code is no longer in the constructor, but in an `initialize` method.

```
package org.globus.examples.services.core.singleton.impl;

import org.globus.wsrf.Resource;
import org.globus.wsrf.ResourceProperties;
import org.globus.wsrf.ResourceProperty;
```

```
import org.globus.wsrf.ResourcePropertySet;
import org.globus.wsrf.impl.SimpleResourcePropertySet;
import org.globus.wsrf.impl.ReflectionResourceProperty;

public class MathResource implements Resource, ResourceProperties {

/* Resource Property set */
private ResourcePropertySet propSet;

/* Resource properties */
private int value;

private String lastOp;

/* Initializes RPs */
public void initialize() throws Exception {
this.propSet = new SimpleResourcePropertySet(
MathQNames.RESOURCE_PROPERTIES);

try {
ResourceProperty valueRP = new ReflectionResourceProperty(
MathQNames.RP_VALUE, "Value", this);
this.propSet.add(valueRP);
setValue(0);

ResourceProperty lastOpRP = new ReflectionResourceProperty(
MathQNames.RP_LASTOP, "LastOp", this);
this.propSet.add(lastOpRP);
setLastOp("NONE");
} catch (Exception e) {
throw new RuntimeException(e.getMessage());
}
}

/* Get/Setters for the RPs */
public int getValue() {
return value;
}

public void setValue(int value) {
this.value = value;
}

public String getLastOp() {
return lastOp;
}

public void setLastOp(String lastOp) {
this.lastOp = lastOp;
}

/* Required by interface ResourceProperties */
public ResourcePropertySet getResourcePropertySet() {
```

```
return this.propSet;
}
}
```

> **Note:** This file is `$EXAMPLES_DIR/org/globus/examples/services/core/singleton/impl/MathResource.java`.

As you can see, this class contains all the code required to implement the resource and its resource properties (`Value` and `LastOp`). Remember that this is the *stateful* component.

The service implementation

This class, on the other hand, will contain the `add`, `subtract`, and `getValueRP` methods the client will interact with. However, we can't reuse the code from the previous chapter. Remember, for example, how we implemented the `add` operation:

```
public AddResponse add(int a) throws RemoteException {
value += a;
lastOp = "ADDITION";

return new AddResponse();
}
```

Now that we're splitting up the implementation, the stateful information is no longer in the service class itself, so we don't have a `value` or `lastOp` variable we can interact with. Any operation that requires working with stateful information will have to work with a `MathResource` object. So, our `add` method will look something like this:

```
public AddResponse add(int a) throws RemoteException {
MathResource mathResource = getResource(); ❶
mathResource.setValue(mathResource.getValue() + a); ❷
mathResource.setLastOp("ADDITION"); ❸

return new AddResponse();
}
```

❶ First of all, we have to get a reference to the resource. The `getResource` method is described shortly.

❷ Now that we have a hold of the resource, we can work with its stateful information. In this step, we simply retrieve the value (using the `getValue` method in the `MathResource` object), add parameter `a`, and set the new value with `setValue`.

❸ Finally, we set the last operation to be "ADDITION".

The `getResource` method used above is a private method that retrieves this service's singleton resource. As you can see, this method simply uses `ResourceContext` (a Globus class) to obtain the resource.

```
private MathResource getResource() throws RemoteException {
Object resource = null;
try {
resource = ResourceContext.getResourceContext().getResource();
} catch (Exception e) {
throw new RemoteException("", e);
}

MathResource mathResource = (MathResource) resource;
return mathResource;
}
```

The complete source code for the service implementation is:

```
package org.globus.examples.services.core.singleton.impl;

import java.rmi.RemoteException;

import org.globus.examples.services.core.singleton.impl.MathResource;
import org.globus.wsrf.ResourceContext;
import org.globus.examples.stubs.MathService_instance.AddResponse;
import org.globus.examples.stubs.MathService_instance.SubtractResponse;
import org.globus.examples.stubs.MathService_instance.GetValueRP;

public class MathService {

/*
 * Private method that gets a reference to the resource specified in the
 * endpoint reference.
 */
private MathResource getResource() throws RemoteException {
Object resource = null;
try {
resource = ResourceContext.getResourceContext().getResource();
} catch (Exception e) {
throw new RemoteException("Unable to access resource.", e);
}

MathResource mathResource = (MathResource) resource;
return mathResource;
}

/* Implementation of add, subtract, and getValue operations */

public AddResponse add(int a) throws RemoteException {
MathResource mathResource = getResource();
mathResource.setValue(mathResource.getValue() + a);
mathResource.setLastOp("ADDITION");

return new AddResponse();
}

public SubtractResponse subtract(int a) throws RemoteException {
```

```
MathResource mathResource = getResource();
mathResource.setValue(mathResource.getValue() - a);
mathResource.setLastOp("SUBTRACTION");

return new SubtractResponse();
}

public int getValueRP(GetValueRP params) throws RemoteException {
MathResource mathResource = getResource();

return mathResource.getValue();
}

}
```

> **Note:** This file is `$EXAMPLES_DIR/org/globus/examples/services/core/singleton/impl/MathService.java`.

The resource home

The implementation of the resource home is extremely simple since we're extending from an existing class included in the toolkit, `SingletonResourceHome`. That base class provides practically all the functionality our resource home needs to manage a single resource. The only thing we have to do is implement the `findSingleton` method, which is called internally by the `ResourceContext` class when we first request the resource. The `findSingleton` method creates a new `MathResource` object, initializes it, and returns it. The base class `SingletonResourceHome` keeps a copy of that resource, which is returned each time the resource is requested.

The complete source code would be:

```
package org.globus.examples.services.core.singleton.impl;

import org.globus.wsrf.Resource;
import org.globus.wsrf.impl.SingletonResourceHome;

public class MathResourceHome extends SingletonResourceHome {

public Resource findSingleton() {
try {
// Create a resource and initialize it.
MathResource mathResource = new MathResource();
mathResource.initialize();
return mathResource;
} catch (Exception e) {
e.printStackTrace();
return null;
}
```

```
}
}
```

Note: This file is `$EXAMPLES_DIR/org/globus/examples/services/core/singleton/impl/MathResourceHome.java`.

The usefulness of having a resource home might not be apparent in this simple case (when we have a singleton resource). However, a resource home adds a lot of versatility to our implementation, specially when we have to deal with multiple resources. For example, a resource home can be used to perform special actions when a resource is created and later destroyed (adding an entry in a database, writing out a log message, etc.). In the next chapter, we will see how we modify the implementation of the resource home to accomodate multiple resources.

Build, deploy, and try it out... with the same client

After splitting up the implementation, we are now ready to build and deploy our new service. First off, the WSDD file has only two small changes: a new service name, and a new service class name. Since we are reusing the WSDL file from the previous chapter, we don't have to change anything else in the WSDD file.

```
<?xml version="1.0" encoding="UTF-8"?>
<deployment name="defaultServerConfig"
    xmlns="http://xml.apache.org/axis/wsdd/"
    xmlns:java="http://xml.apache.org/axis/wsdd/providers/java"
    xmlns:xsd="http://www.w3.org/2001/XMLSchema">

    <service name="examples/core/singleton/MathService" provider="Handler" use="literal" st
        <parameter name="className" value="org.globus.examples.services.core.singleton.impl
        <wsdlFile>share/schema/examples/MathService_instance/Math_service.wsdl</wsdlFile>
        <parameter name="allowedMethods" value="*"/>
        <parameter name="handlerClass" value="org.globus.axis.providers.RPCProvider"/>
        <parameter name="scope" value="Application"/>
        <parameter name="providers" value="GetRPProvider"/>
        <parameter name="loadOnStartup" value="true"/>
    </service>

</deployment>
```

Note: This file is `$EXAMPLES_DIR/org/globus/examples/services/core/singleton/deploy-server.wsdd`.

Now, let's take a closer look at the JNDI deployment file. In the previous chapter we asked you to please take a leap of faith and accept that we needed this file "because we need it". Now, we can explain what this file does. It is responsible for specifying what resource home our service has to use to get a hold of

resources. In this file we will also specify parameters related to how the resource home manages those resources. However, since at this point we are only managing a single resource, our JNDI deployment file will be pretty simple.

```
<?xml version="1.0" encoding="UTF-8"?>
<jndiConfig xmlns="http://wsrf.globus.org/jndi/config">

<service name="examples/core/singleton/MathService">
<resource name="home" type="org.globus.examples.services.core.singleton.impl.MathResourceHo
<resourceParams>

<parameter>
<name>factory</name>
<value>org.globus.wsrf.jndi.BeanFactory</value>
</parameter>

</resourceParams>

</resource>
</service>

</jndiConfig>
```

> **Note:** This file is `$EXAMPLES_DIR/org/globus/examples/services/core/singleton/deploy-jndi-config.xml`.

In a nutshell, the root element `jndiConfig` of the file can contain several `service` elements (one for each service we're configuring... since we're only configuring one service, we'll have a single `service` element). This element has a `name` attribute whose value *must match* the service name specified in the WSDD file (this is basically the 'glue' between the WSDD file and the JNDI file).

The `service` element contains a `resource` element which we'll use to specify the resource home for our service. Notice that we do so using the `type` attribute. The `resource` element can contain several resource parameters. In this example we have a single parameter, which will be common to all the services we deploy.

> **Note:** If you look back on the JNDI deployment file from the previous chapter, you'll see that it is the same as the one shown above, except that the resource home we specify is `ServiceResourceHome`.

Now, we can build the service:

```
./globus-build-service.sh singleton
```

And deploy it:

```
globus-deploy-gar $EXAMPLES_DIR/org_globus_examples_services_core_singleton.gar
```

> **Note:** You will need to restart the Globus standalone container for the deployment to take effect.

Finally, we're going to make sure the service works. Another nice perk of reusing the WSDL file is that we can also reuse the client described in the previous chapter. Remember that we've only changed the implementation of the service; as long as we don't change the WSDL interface, there's no need to write a new client.

So, remember you have to run the client like so:

```
java \
-classpath ./build/stubs/classes/:$CLASSPATH \
org.globus.examples.clients.MathService_instance.Client \
http://127.0.0.1:8080/wsrf/services/examples/core/singleton/MathService
```

If all goes well, you should see the following:

```
Current value: 15
Current value: 10
```

If you run the client another time, since our service is tied to a singleton resource, you should see the value of increase with each run of the client.

```
Current value: 25
Current value: 20

Current value: 35
Current value: 30
```

Chapter 5. Multiple resources

In the previous two chapters we implemented a simple stateful web service that used a single resource to keep stateful information. First, we used the `ServiceResourceHome` so we could implement the service and the resource in the same class, and then we split up the implementation into a service class, a resource class, and a resource home class.

In this chapter we will learn how to write a service that, using a design pattern known as the *factory/instance* pattern, will be able to manage *multiple* resources.

The WS-Resource factory pattern

The factory/instance patterns is a well-known design pattern in software design, and specially in object-oriented languages. In this pattern, we are not allowed to create instances of objects directly, but must do so through a *factory* that will provide a `create` operation.

When dealing with multiple resources, the WSRF specs recommend that we follow this pattern, having one service in charge of creating the resources (*"the factory service"*) and another one to actually access the information contained in the resources (*"the instance service"*).

Figure 5-1. The WS-Resource factory pattern

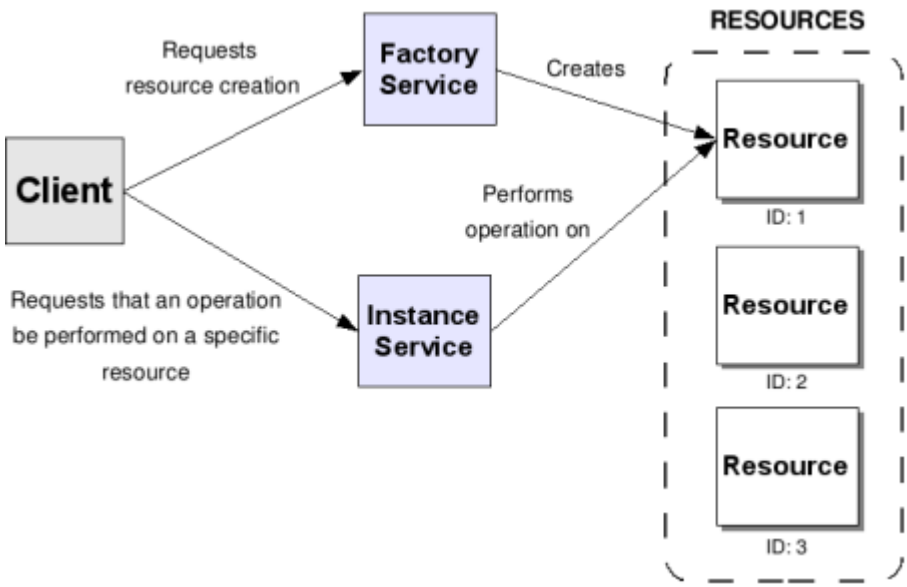


Figure 5-1 summarizes the relationship between these two services, the resources, and the client. Whenever the client wants to create a new resource, it will contact the factory service, who will take care of creating and initializing a new resource. It is important to see that, in this case, the resource is also assigned *a unique key*. Since we are no longer dealing with a single resource, we need some way of telling each resource apart. The factory service will return an *endpoint reference* to a WS-Resource composed of the instance service and the recently created resource.

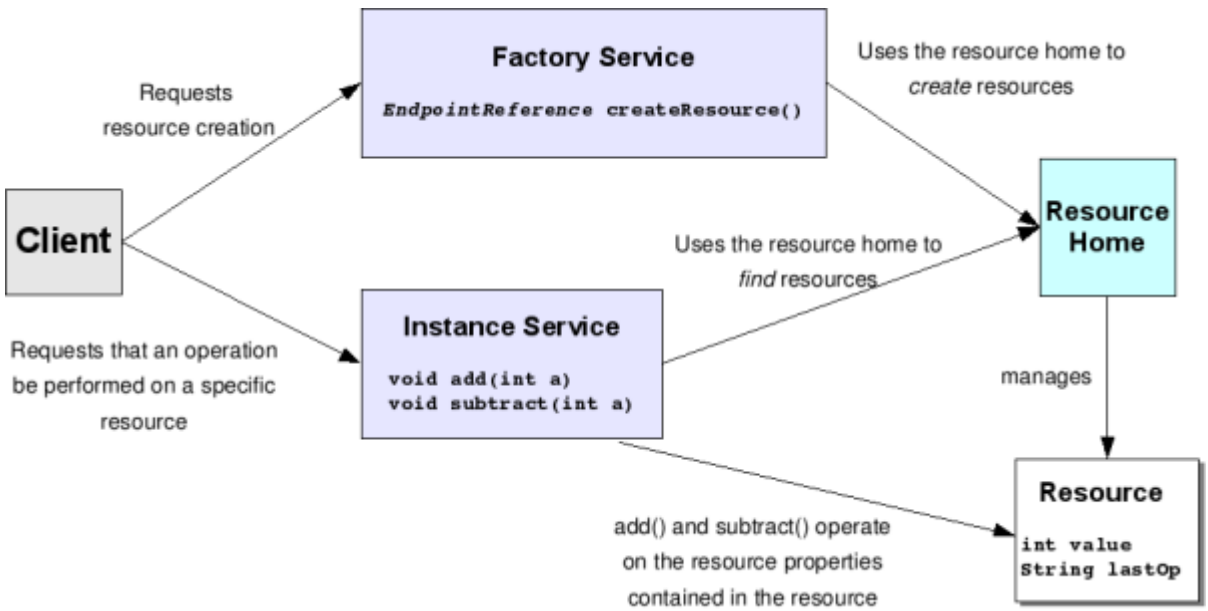
> **Note:** Remember from that endpoint references are a part of the WS-Addressing specification. Endpoint references (or EPR's for short) allow us to uniquely address a single WS-Resource. Also, remember that a *WS-Resource* is the pairing of a service with a particular resource. In our first example client, our EPR included only the service's URI (because we had a single resource). In this chapter, our EPR's will have both the service's URI and the resource's key.

Using the EPR returned by the factory, the client can now invoke the service's operations through the instance service. This service, in turn, will perform the operations using the recently created resource.

Implementing the WS-Resource factory pattern in GT4

Implementing this design pattern in GT4 is actually not too complicated. In fact, it is very similar to the example seen in the previous chapter, with two main differences, highlighted in Figure 5-2 (compare with)

Figure 5-2. Relationships between the Factory Service, the Instance Service, the Resource Home, and the Resource



- **Factory service and instance service**. To handle multiple resources, we will now need to deploy two services: a factory service and an instance service. The factory service provides a `createResource` operation that returns an EPR to the new WS-Resource. The instance service provides the operations we have been working with in the previous chapters: `add`, `subtract`, and `getValueRP`.
- **Non-trivial resource home**. The resource home no longer deals with a single resource. In this case, it must keep track of several resources at the same time. However, notice how we have a single resource

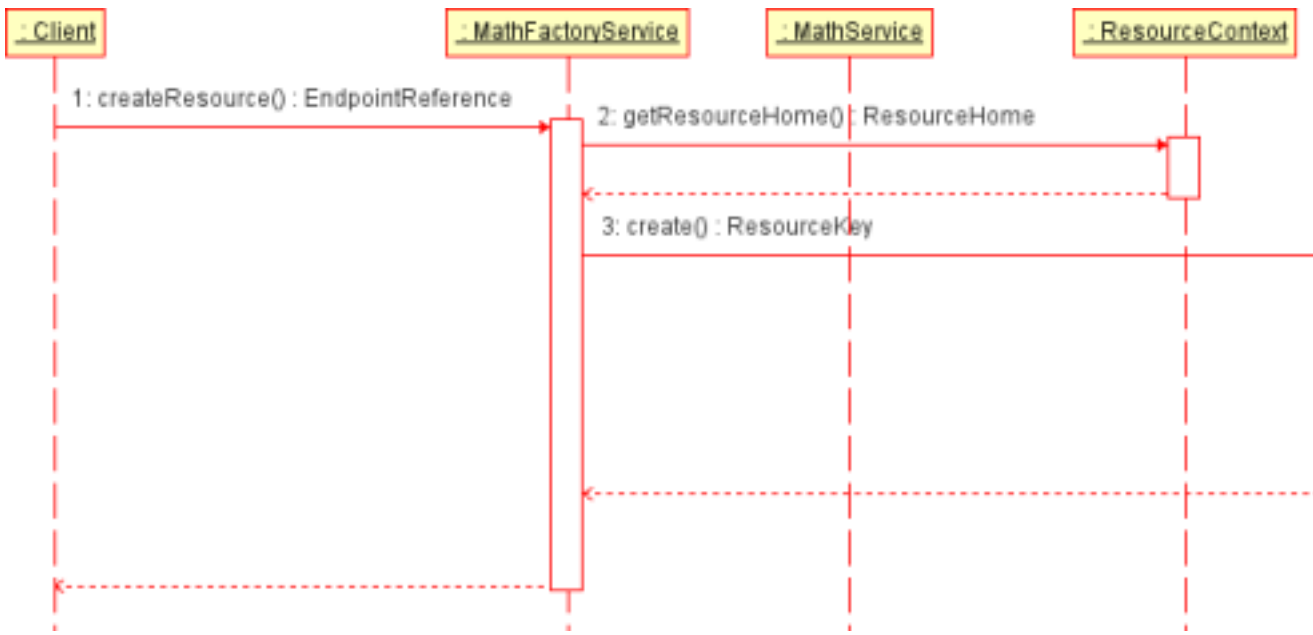
home, shared by the factory and instance services. The factory service will use the resource home to *create* new resources, while the instance service will use it to *find* a resource with a given key.

So, for this example, we will have four Java classes (plus the `MathQNames` interface)

- The factory service:
  `$EXAMPLES_DIR/org/globus/examples/services/core/factory/impl/MathFactoryService.java`
- The instance service:
  `$EXAMPLES_DIR/org/globus/examples/services/core/factory/impl/MathService.java`
- The resource:
  `$EXAMPLES_DIR/org/globus/examples/services/core/factory/impl/MathResource.java`
- The resource home:
  `$EXAMPLES_DIR/org/globus/examples/services/core/factory/impl/MathResourceHome.java`

Before seeing the actual code, a good way of seeing what role each class plays is to see how they interact. For now, don't worry about all the deployment details. Just imagine that we actually have our two services (factory and instance) up and ready to accept invocations from a client class, and that the service class has access to the resource home and that the resource home, in turn, has access to a bunch of resource objects. Let's start with the creation of a new resource, shown in Figure 5-3.

Figure 5-3. Sequence diagram for resource creation



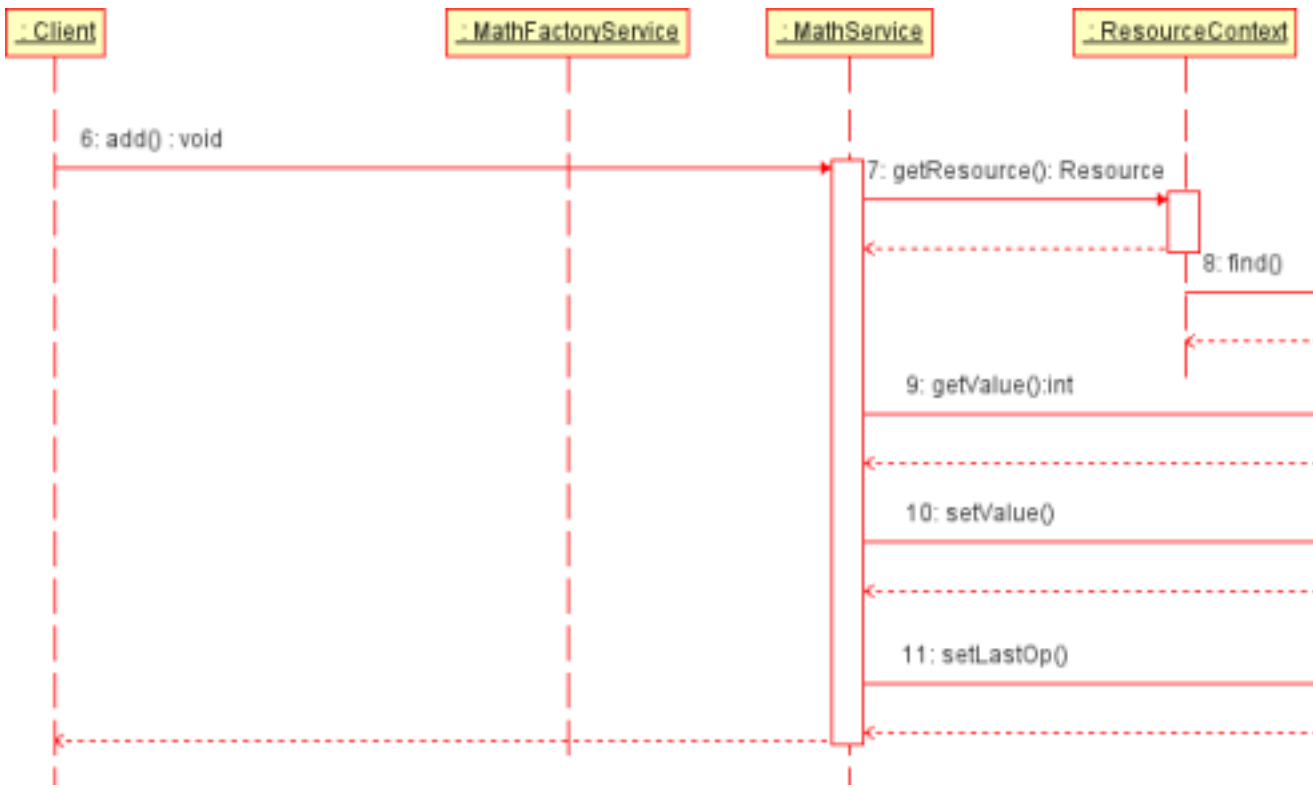
1. Our client only needs to know the URI of the factory service (`MathFactoryService`). With it, it can invoke the `createResource` operation. This will return an *endpoint reference* containing the URI of the instance service, along with the key of the recently created resource.
2. So, the factory service has to create a new resource. This necessarily has to be done through the resource home, which is in charge of managing all the resources. However, we have to locate our

resource home first. Fortunately, this is quite easy since we can delegate this task on a Globus-supplied class called `ResourceContext`. In the previous chapter, we used this class to retrieve the service's singleton resource. It can also be used to obtain a reference to the service's resource home.

3. Now that we have the resource home, we can ask it to create the new resource. The creation method returns an object of type `ResourceKey`. This is the resource identifier which we need to create the endpoint reference we'll be returning to the client.
4. The resource home will take care of actually creating a new `MathResource` instance.
5. Next, the resource home will add the new `MathResource` instance to its internal list of resources. This list will allow us to access any resource if we know that resource's identifier.

Once the `createResource` call has finished, the client will have the WS-Resource's endpoint reference. In all future calls, this endpoint reference will be passed along *transparently* in all our invocations. In other words, when we call `add` or `subtract`, the service class will know what resource we're referring to. So, let's take a close look at what happens when we invoke the `add` operation, as shown in Figure 5-4.

Figure 5-4. Sequence diagram for WS-Resource invocation



6. The client invokes the `add` operation in the instance service (`MathService`).
7. However, the `add` operation is stateless. It needs to retrieve a resource to actually work. The resource identifier is in the endpoint reference that is included in the invocation. Fortunately, the

`ResourceContext` helper class once again shields us from all the potential nastiness. It will be in charge of reading the EPR and finding the resource it refers to.

8. However, it's interesting to note that, internally, *ResourceContext* uses the *ResourceHome* to find the resource.
9. Once we have the resource, the instance service can access all its state information, such as the "Value" and the "LastOp" resource properties. First of all, we will access the "Value" resource property. As in the example seen in the previous chapter, our resource (`MathResource`) will allow us to modify the RP's using get/set methods (in this case, with a simple `getValue()` method).
10. Once we've modified the value, we have to make sure we commit the change in the resource (in our case, using `setValue()`). Otherwise, that bit of state information won't be remembered.
11. Finally, we use the `setLastOp()` method in the resource to modify the `LastOp` resource property to equal `"ADDITION"`.

Don't worry if you're a bit confused. When we actually start coding all this, it'll probably seem clearer (even so, you might want to review these diagrams once we've coded the full example).

The factory service

We begin by implementing the factory service. The WSDL file for the factory service is very simple, as we only have a single operation `createResource` with no parameters and that returns an endpoint reference.

> **Note:** you should be able to read the file and recognize that there is, indeed, a single operation `createResource` with no parameters and returning an endpoint reference. If not, this might be a good time to review .

```
<?xml version="1.0" encoding="UTF-8"?>
<definitions name="FactoryService"
    targetNamespace="http://www.globus.org/namespaces/examples/core/FactoryService" ❶
    xmlns="http://schemas.xmlsoap.org/wsdl/"
    xmlns:tns="http://www.globus.org/namespaces/examples/core/FactoryService"
    xmlns:wsa="http://schemas.xmlsoap.org/ws/2004/03/addressing" ❷
    xmlns:wsdl="http://schemas.xmlsoap.org/wsdl/"
    xmlns:xsd="http://www.w3.org/2001/XMLSchema">


<!--============================================================

                    T Y P E S

  ============================================================-->
<types>
<xsd:schema targetNamespace="http://www.globus.org/namespaces/examples/core/FactoryService"
    xmlns:tns="http://www.globus.org/namespaces/examples/core/FactoryService"
    xmlns:xsd="http://www.w3.org/2001/XMLSchema">
```

```
❸
<xsd:import
    namespace="http://schemas.xmlsoap.org/ws/2004/03/addressing"
    schemaLocation="../../ws/addressing/WS-Addressing.xsd" />


<!-- REQUESTS AND RESPONSES -->

<xsd:element name="createResource">
<xsd:complexType/>
</xsd:element>
<xsd:element name="createResourceResponse">
<xsd:complexType>
<xsd:sequence>
<xsd:element ref="wsa:EndpointReference"/>
</xsd:sequence>
</xsd:complexType>
</xsd:element>

</xsd:schema>
</types>


<!--============================================================

                    M E S S A G E S

  ============================================================-->
<message name="CreateResourceRequest">
<part name="request" element="tns:createResource"/>
</message>
<message name="CreateResourceResponse">
<part name="response" element="tns:createResourceResponse"/>
</message>



<!--============================================================

                    P O R T T Y P E

  ============================================================-->
❹
<portType name="FactoryPortType">

<operation name="createResource">
<input message="tns:CreateResourceRequest"/>
<output message="tns:CreateResourceResponse"/>
</operation>

</portType>
```

```
</definitions>
```

Note: This file is `$EXAMPLES_DIR/schema/examples/FactoryService/Factory.wsdl`.

❶ The factory service's target namespace is `http://www.globus.org/namespaces/examples/core/FactoryService`.

❷ The service's `createResource` operation returns an endpoint reference, a structure that is part of the WS-Addressing specification. We need to declare the WS-Addressing namespace.

❸ We also need to import the WS-Addressing Schema file, which contains the definition of the endpoint reference structure (`wsa:EndpointReference`)

❹ Our portType, `FactoryPortType`, has a single operation `createResource`.

Since we have added a new interface, we need to map the new WSDL namespaces to Java packages (as described in)

```
http\://www.globus.org/namespaces/examples/core/FactoryService=
                                    org.globus.examples.stubs.Factory
http\://www.globus.org/namespaces/examples/core/FactoryService/bindings=
                                    org.globus.examples.stubs.Factory.bindings
http\://www.globus.org/namespaces/examples/core/FactoryService/service=
                                    org.globus.examples.stubs.Factory.service
```

Note: These three lines must be present in `$EXAMPLES_DIR/namespace2package.properties`.

Now, we have to write the Java implementation of the factory service. This will be a single Java class, with a single `createResource` method:

```
public class MathFactoryService {

/* Implementation of createResource Operation */
public CreateResourceResponse createResource(CreateResource request)
throws RemoteException {

}
}
```

Note: This is part of file `$EXAMPLES_DIR/org/globus/examples/services/core/factory/impl/MathFactoryService.java`.

Inside this method, we will perform three steps:

1. Retrieve the resource home.
2. Use the resource home to create a new resource.
3. Create the endpoint reference we will return to the client. This EPR must contain the instance service's URI and the new resource's key.

In the following snippet of code we perform steps 1 and 2:

```
ResourceContext ctx = null;
MathResourceHome home = null;
ResourceKey key = null;
try {
ctx = ResourceContext.getResourceContext();
home = (MathResourceHome) ctx.getResourceHome();
key = home.create();
} catch (Exception e) {
throw new RemoteException("", e);
}
```

If we succeed in retrieving the resource home, then the `key` variable will contain the resource's identifier, which we'll use to construct the endpoint reference:

```
EndpointReferenceType epr = null;

try {
URL baseURL = ServiceHost.getBaseURL();
String instanceService = (String) MessageContext
.getCurrentContext().getService().getOption("instance");
String instanceURI = baseURL.toString() + instanceService;
// The endpoint reference includes the instance's URI and the resource key
epr = AddressingUtils.createEndpointReference(instanceURI, key);
} catch (Exception e) {
throw new RemoteException("", e);
}
```

Notice how we have to create a new endpoint reference (of type `EndpointReferenceType`) using the instance server's URI (`instanceURI`) and the resource's identifier (`key`). Finally, the only thing left to do is to 'box' the EPR inside a `CreateResourceResponse` object and return it.

```
CreateResourceResponse response = new CreateResourceResponse();
response.setEndpointReference(epr);
return response;
```

> **Note:** Remember from the information box in that whenever one of our operations returns a complex type (such as an endpoint reference), it is 'boxed' inside a stub class.

> **Note:** Now is a good moment to review Figure 5-3.

The instance service

Implementing the instance service is going to be very simple, since we can reuse both the WSDL file and practically all the Java implementation from the previous chapter (with only minor changes to the resource class).

> **Note:** The implementation of the instance service is `$EXAMPLES_DIR/org/globus/examples/services/core/factory/impl/MathService.java`.

But let's not leave it at that: this is a good moment to ask ourselves: why exactly can we reuse both the WSDL file and the Java implementation of the service and the resource? Well, take into account that what we are doing in this chapter, in a sense, is a *generalization* of the singleton service. In the previous chapter, we were interested in having a service with operations `add`, `subtract`, and `getValueRP` that interacted with a single (or *singleton*) resource. Interacting with *multiple* resources doesn't fundamentally affect the implementation of the service that will be accessing the resources; it only affects those parts of our application that are in charge of *managing* the resource. We've added a new factory service and, as we shall see right now, we have to modify the resource home. But the instance service is unaffected because it doesn't really care whether it has access to one or a million resources.

Even so, this doesn't mean that there isn't more stuff happening in the 'background' in the instance service. For example, in the `add` operation:

```
public AddResponse add(int a) throws RemoteException {
MathResource mathResource = getResource();
mathResource.setValue(mathResource.getValue() + a);
mathResource.setLastOp("ADDITION");

return new AddResponse();
}
```

We are still calling the `getResource`, which is still implemented as in the previous chapter:

```
private MathResource getResource() throws RemoteException {
Object resource = null;
try {
resource = ResourceContext.getResourceContext().getResource();
} catch (Exception e) {
throw new RemoteException("", e);
}

MathResource mathResource = (MathResource) resource;
return mathResource;
}
```

However, the Globus-supplied `ResourceContext` will do more than simply look up a singleton resource. It will look inside the endpoint reference that is used to invoke `add`, extract the resource key, and lookup the corresponding resource through the resource home. One of the nice things about using endpoint references, instead of plain URIs, is that the resource key is passed to the service *transparently*,

so our methods can be declared simply as `add(int a)`, instead of something like `add(int a, int resourceID)`.

> **Note:** Now is a good moment to review Figure 5-4.

The resource

The resource implementation requires only minimal changes. Remember that each resource will now have a unique key identifying it. We will need to reflect this in our resource by implementing the *ResourceIdentifier* interface:

```
public class MathResource implements Resource, ResourceIdentifier,
ResourceProperties {

}
```

This interface requires us to implement a *getID* method returning the resource's identifier. Notice that the identifier is of type `Object`. In other words, our unique identifier can be of any type we wish, although we will generally choose a key of type `Integer`. Later on we will see that we will need to specify the type of the resource identifier in the JNDI deployment file.

```
/* Resource key. This uniquely identifies this resource. */
private Object key;

/* Required by interface ResourceIdentifier */
public Object getID() {
return this.key;
}
```

The initialization of the resource identifier takes place in the `initialize` method of the resource class. The simplest key we can create is the resource's hashCode (in Java, every object has an identifier which can be retrieved by calling the `hashCode` method). Notice how the `initialize` method now returns the resource identifier.

```
/* Initializes RPs and returns a unique identifier for this resource */
public Object initialize() throws Exception {
this.key = new Integer(hashCode());

// Initialize the resource properties

return key;
}
```

> **Note:** This is part of file `$EXAMPLES_DIR/org/globus/examples/services/core/factory/impl/MathResource.java`.

> **Warning**
>
> An object's `hashCode` isn't guaranteed to be unique. For the simple examples included in the tutorial, this should not be a problem. However, using other unique identifiers is prefered for more complex services, specially if our service will work with persistent resources. For example, you can use the `UUIDGen` class included with Apache Axis.

The resource home

The resource home has to be modified, but still remains relatively simple because, as in the previous chapter, our resource class extends from a Globus-supplied class. In the previous chapter, we extended from `SingletonResourceHome`, a Globus-supplied class that provided most of the functionality of a resource home for a single resource. Now we will extend from `ResourceHomeImpl`, another Globus-supplied class for resource homes that manage several resources.

The only method we have to implement is the `create` method, where we will create a new resource and return its identifier. All the other methods we would expect in a resource home (such as a `find` method to retrieve a resource given a certain key) are already implemented for us in `ResourceHomeImpl`.

```
package org.globus.examples.services.core.factory.impl;

import org.globus.wsrf.ResourceKey;
import org.globus.wsrf.impl.ResourceHomeImpl;
import org.globus.wsrf.impl.SimpleResourceKey;

public class MathResourceHome extends ResourceHomeImpl {

public ResourceKey create() throws Exception {
// Create a resource and initialize it
MathResource mathResource = (MathResource) createNewInstance(); ❶
mathResource.initialize(); ❷
// Get key
ResourceKey key = new SimpleResourceKey(keyTypeName, mathResource
.getID()); ❸
// Add the resource to the list of resources in this home
add(key, mathResource); ❹
return key; ❺
}

}
```

Note: This file is `$EXAMPLES_DIR/org/globus/examples/services/core/factory/impl/MathResourceHome.java`.

❶ We create a new instance of the resource. Notice that this *must* be done using the protected `createNewInstance` method, *not* by using the `new` operator. Also, since `createNewInstance` returns a `Resource` object, we must cast it to our specific resource type: `MathResourceType`

❷ We initialize the resource.

❸ We obtain the resource identifier using the `getID` method implemented earlier, and use it to create a `SimpleResourceKey` object. When creating the `SimpleResourceKey`, `keyTypeName` is a protected attribute of `ResourceHomeImpl` containing the key's type.

❹ We add the recently created resource and its key to the resource home's internal list of resources. `add` is a protected method of `ResourceHomeImpl`.

❺ Finally, we return the resource's key.

Note: Now is *another* good moment to review Figure 5-3.

There's more to the resource home that meets the eye...: The resource home shown above, along with the one seen in the previous chapter, covers the simplest possible case of resources: *in-memory resources*, or resources which reside in main memory while the container is running. However, resource homes can also be used to manage *persistent resources*, or resources that are stored in disk so they can survive container restarts. To do this, our resource must implement the `PersistenceCallback` interface. More details are available in the official Globus documentation.

Another thing we can do to a resource home is to override its `add` and `remove` methods, to control exactly what happens when a resource is added or removed from the resource home. For example, we might want to write to a log, or register our resource with an index service.

Build and deploy

The deployment descriptor

The WSDD file must now reflect that we have two services: the factory service and the instance service.

```
<?xml version="1.0" encoding="UTF-8"?>
<deployment name="defaultServerConfig"
    xmlns="http://xml.apache.org/axis/wsdd/"
    xmlns:java="http://xml.apache.org/axis/wsdd/providers/java"
    xmlns:xsd="http://www.w3.org/2001/XMLSchema">

    ❶
    <!-- Instance service -->
    <service name="examples/core/factory/MathService" provider="Handler" use="literal" styl
        <parameter name="className" value="org.globus.examples.services.core.factory.impl.M
        <wsdlFile>share/schema/examples/MathService_instance/Math_service.wsdl</wsdlFile>
```

```
        <parameter name="allowedMethods" value="*"/>
        <parameter name="handlerClass" value="org.globus.axis.providers.RPCProvider"/>
        <parameter name="scope" value="Application"/>
        <parameter name="providers" value="GetRPProvider"/>
    </service>

    ❷
    <!-- Factory service -->
    <service name="examples/core/factory/MathFactoryService" provider="Handler" use="litera
        <parameter name="className" value="org.globus.examples.services.core.factory.impl.M
        <wsdlFile>share/schema/examples/FactoryService/Factory_service.wsdl</wsdlFile>
        <parameter name="allowedMethods" value="*"/>
        <parameter name="handlerClass" value="org.globus.axis.providers.RPCProvider"/>
        <parameter name="scope" value="Application"/>
        <parameter name="instance" value="examples/core/factory/MathService"/> ❸
    </service>

</deployment>
```

Note: This file is `$EXAMPLES_DIR/org/globus/examples/services/core/factory/deploy-server.wsdd`.

❶ The parameters for the instance service are the same as the ones used in the previous chapter.

❷ The deployment parameters of the factory are pretty straightforward. We simply specify the factory service's class (`MathFactoryService`) and WSDL file.

❸ The only new parameter is the `instance` parameter, which must be set to the `name` of the instance service (as written in its `<service>` tag).

The JNDI deployment file

In the previous chapter we saw that this file specifies what resource home must be used by each service. When managing just one resource, this file was very simple. Now, however, we will need to specify more parameters to manage multiple resources. Furthermore, our JNDI deployment file must include two `<service>` tags (one for the instance service, and one for the factory service).

```
<?xml version="1.0" encoding="UTF-8"?>
<jndiConfig xmlns="http://wsrf.globus.org/jndi/config">

<!-- Instance service -->
<service name="examples/core/factory/MathService">
<resource name="home" type="org.globus.examples.services.core.factory.impl.MathResourceHome
<resourceParams>
```

```
❷
<parameter>
<name>resourceClass</name>
<value>org.globus.examples.services.core.factory.impl.MathResource</value>
</parameter>

❸
<parameter>
<name>resourceKeyType</name>
<value>java.lang.Integer</value>
</parameter>

❹
<parameter>
<name>resourceKeyName</name>
<value>{http://www.globus.org/namespaces/examples/core/MathService_instance}MathResourceKey
</parameter>

❺
<parameter>
<name>factory</name>
<value>org.globus.wsrf.jndi.BeanFactory</value>
</parameter>

</resourceParams>
</resource>
</service>

<!-- Factory service -->
<service name="examples/core/factory/MathFactoryService">
❻
<resourceLink name="home" target="java:comp/env/services/examples/core/factory/MathService/
</service>

</jndiConfig>
```

> **Note:** This file is `$EXAMPLES_DIR/org/globus/examples/services/core/factory/deploy-jndi-config.xml`.

❶ We use the `<resource>` tag to specify what resource home will be used by the instance service.

❷ The `resourceClass` parameter specifies the type of our resources.

❸ The value we put in the `resourceKeyType` parameter must match the type used in the implementation when creating the key. Remember that we used the resource's hash code, which is of type `java.lang.Integer`. Thus, we have to include that type in the JNDI configuration.

❹ The `resourceKeyName` parameter must be a qualified name. Notice that we're using a name that is not mentioned in our WSDL file. This is Ok and, in fact, we could use any QName we wanted. However, for clarity we should choose a QName that is in the same namespace as the service.

❺ Remember from the previous chapter that this is a common parameter which we will find in all our JNDI deployment files. However, it is important to note that, even though this parameter is called `factory`, it has nothing to do with the factory/instance pattern we are seeing in this chapter. It refers to a completely different factory within the Globus code.

❻ The factory service uses *the same resource home* as the instance service. So, we do not need to repeat all the parameters of the instance service. We simply have to include a `<resourceLink>` tag linking to the previously specified resource home. Notice that we must do so using a special path.

Build and deploy

We are finally ready to build and deploy our service:

```
./globus-build-service.sh factory
```

```
globus-deploy-gar $EXAMPLES_DIR/org_globus_examples_services_core_factory.gar
```

A simple client

We will try out our service first with a simple client that creates a new resource and performs a couple operations on it. This client expects only one argument from the command line, the factory service's URI

```
package org.globus.examples.clients.FactoryService_Math;

import org.apache.axis.message.addressing.Address;
import org.apache.axis.message.addressing.EndpointReferenceType;

import org.globus.examples.stubs.MathService_instance.GetValueRP;
import org.globus.examples.stubs.MathService_instance.MathPortType;
import org.globus.examples.stubs.MathService_instance.service.MathServiceAddressingLocator;
import org.globus.examples.stubs.Factory.service.FactoryServiceAddressingLocator;
import org.globus.examples.stubs.Factory.FactoryPortType;
import org.globus.examples.stubs.Factory.CreateResource;
import org.globus.examples.stubs.Factory.CreateResourceResponse;

/* This client creates a new MathService instance through a FactoryService. This client
 * expects one parameter: the factory URI.
 */
public class Client {

public static void main(String[] args) {
FactoryServiceAddressingLocator factoryLocator = new FactoryServiceAddressingLocator();
MathServiceAddressingLocator instanceLocator = new MathServiceAddressingLocator();

try {
String factoryURI = args[0];
```

```
EndpointReferenceType factoryEPR, instanceEPR;
FactoryPortType mathFactory;
MathPortType math;

❶
factoryEPR = new EndpointReferenceType();
factoryEPR.setAddress(new Address(factoryURI));
mathFactory = factoryLocator.getFactoryPortTypePort(factoryEPR);

❷
CreateResourceResponse createResponse = mathFactory
.createResource(new CreateResource());
instanceEPR = createResponse.getEndpointReference();

❸
math = instanceLocator.getMathPortTypePort(instanceEPR);

System.out.println("Created instance.");

❹
// Perform an addition
math.add(10);

// Perform another addition
math.add(5);

// Access value
System.out
.println("Current value:" + math.getValueRP(new GetValueRP()));

// Perform a subtraction
math.subtract(5);

// Access value
System.out
.println("Current value:" + math.getValueRP(new GetValueRP()));
} catch (Exception e) {
e.printStackTrace();
}
}

}
```

Note: This file is `$EXAMPLES_DIR/org/globus/examples/clients/FactoryService_Math/Client.java`.

❶ Here we obtain a reference to the factory's portType. Notice how we only need the factory's URI to do this.

❷ Once we have the factory's portType, we use it to invoke the `createResource` operation. This operation returns an endpoint reference, 'boxed' inside a `CreateResourceResponse` object. This endpoint reference includes both the instance service's URI *and* the new resource's identifier. In the next client we will take a peek inside the endpoint reference.

❸ Using the instance EPR, we obtain a reference to the `MathPortType` in the instance service.

❹ We now use the `MathPortType` to invoke `add`, `subtract`, and `getValueRP`.

Compile and run the client:

```
javac \
-classpath ./build/stubs/classes/:$CLASSPATH \
org/globus/examples/clients/FactoryService_Math/Client.java

java \
-classpath ./build/stubs/classes/:$CLASSPATH \
org.globus.examples.clients.FactoryService_Math.Client \
http://127.0.0.1:8080/wsrf/services/examples/core/factory/MathFactoryService
```

If all goes well, you should see the following:

```
Created instance.
Current value:15
Current value:10
```

If you run it again, you will get the exact same result. This is because we are creating a new resource every time we run the client.

```
Created instance.
Current value:15
Current value:10
```

A slightly less simple client

We will now split the previous client into two client applications: a client in charge of creating the resource, and a client in charge of invoking the `add` operation in the instance service. The first client writes the endpoint reference of the new resource to a file, which will later be read by the second client.

The creating client

The first client expects at least one parameter from the command line: the factory service's URI. A second parameter, with the name of the file where the EPR must be written to, can also be specified. If it isn't, then it will be saved to a file called `epr.txt`.

```
package org.globus.examples.clients.FactoryService_Math;

import java.io.BufferedWriter;
```

```
import java.io.FileWriter;

import org.apache.axis.message.addressing.Address;
import org.apache.axis.message.addressing.EndpointReferenceType;
import org.globus.examples.services.core.factory.impl.MathQNames;

import org.globus.examples.stubs.MathService_instance.MathPortType;
import org.globus.examples.stubs.MathService_instance.service.MathServiceAddressingLocator;
import org.globus.examples.stubs.Factory.service.FactoryServiceAddressingLocator;
import org.globus.examples.stubs.Factory.FactoryPortType;
import org.globus.examples.stubs.Factory.CreateResource;
import org.globus.examples.stubs.Factory.CreateResourceResponse;
import org.globus.wsrf.encoding.ObjectSerializer;

public class ClientCreate {

static final String EPR_FILENAME = "epr.txt";

public static void main(String[] args) {
FactoryServiceAddressingLocator factoryLocator = new FactoryServiceAddressingLocator();
MathServiceAddressingLocator instanceLocator = new MathServiceAddressingLocator();

try {
String factoryURI = args[0];
String eprFilename;

if(args.length==2)
eprFilename=args[1];
else
eprFilename=EPR_FILENAME;

EndpointReferenceType factoryEPR, instanceEPR;
FactoryPortType mathFactory;
MathPortType math;

❶
factoryEPR = new EndpointReferenceType();
factoryEPR.setAddress(new Address(factoryURI));
mathFactory = factoryLocator.getFactoryPortTypePort(factoryEPR);

CreateResourceResponse createResponse = mathFactory
.createResource(new CreateResource());
instanceEPR = createResponse.getEndpointReference();

❷
String endpointString = ObjectSerializer.toString(instanceEPR,
MathQNames.RESOURCE_REFERENCE);
FileWriter fileWriter = new FileWriter(eprFilename);
BufferedWriter bfWriter = new BufferedWriter(fileWriter);
bfWriter.write(endpointString);
bfWriter.close();
System.out.println("Endpoint reference written to file "
+ eprFilename);
```

```
} catch (Exception e) {
e.printStackTrace();
}
}

}
```

Note: This file is `$EXAMPLES_DIR/org/globus/examples/clients/FactoryService_Math/ClientCreate.java`.

❶ As in the previous client, here we obtain a reference to the factory's portType and use it to invoke the `createResource` operation, that returns and endpoint reference to the new resource.

❷ This block of code writes the endpoint reference to a file. We use the Globus-supplied class `ObjectSerializer`, which creates an XML representation of the EPR. Note that we need to specify the QName of the root element of the XML file. We can choose any name we want but, for clarity, it is best to choose a QName inside our service's namespace. The QName we're using is declared in the MathQNames interface:

```
public static final QName RESOURCE_REFERENCE = new QName(NS,
"MathResourceReference");
```

Now compile the client:

```
javac \
-classpath ./build/stubs/classes/:$CLASSPATH \
org/globus/examples/clients/FactoryService_Math/ClientCreate.java

java \
-classpath ./build/stubs/classes/:$CLASSPATH \
org.globus.examples.clients.FactoryService_Math.ClientCreate \
http://127.0.0.1:8080/wsrf/services/examples/core/factory/MathFactoryService
```

If all goes well, you should see the following:

```
Endpoint reference written to file epr.txt
```

Let's take a look inside the `epr.txt` file:

```
<ns1:MathResourceReference xsi:type="ns2:EndpointReferenceType"
    xmlns:ns1="http://www.globus.org/namespaces/examples/core/MathService_instance"
    xmlns:xsi="http://www.w3.org/2001/XMLSchema-instance"
    xmlns:ns2="http://schemas.xmlsoap.org/ws/2004/03/addressing">

    <ns2:Address xsi:type="ns2:AttributedURI">
    http://127.0.0.1:8080/wsrf/services/examples/core/factory/MathService
    </ns2:Address>
```

```
    <ns2:ReferenceProperties xsi:type="ns2:ReferencePropertiesType">
        <ns1:MathResourceKey>24156236</ns1:MathResourceKey>
    </ns2:ReferenceProperties>

    <ns2:ReferenceParameters xsi:type="ns2:ReferenceParametersType"/>

</ns1:MathResourceReference>
```

Notice how the endpoint reference does, in fact, include the instance service's URI and the resource's key. Note that you will almost certainly get a different key in your resource.

The adding client

This client expects two arguments from the command line. The first argument is a service's URI or the name of a file containing and endpoint reference. The client is implemented to recognize both formats since, in the next chapters, we will use this client again to interact with singleton services (where we only need the service's URI, without a resource key, to address the service). The second argument is the value we wish to add.

```
package org.globus.examples.clients.MathService_instance;

import java.io.FileInputStream;

import org.apache.axis.message.addressing.Address;
import org.apache.axis.message.addressing.EndpointReferenceType;

import org.globus.examples.stubs.MathService_instance.GetValueRP;
import org.globus.examples.stubs.MathService_instance.MathPortType;
import org.globus.examples.stubs.MathService_instance.service.MathServiceAddressingLocator;
import org.globus.wsrf.encoding.ObjectDeserializer;
import org.xml.sax.InputSource;

public class ClientAdd {

public static void main(String[] args) {
MathServiceAddressingLocator instanceLocator = new MathServiceAddressingLocator();

try {
int value = Integer.parseInt(args[1]);
EndpointReferenceType instanceEPR;

if (args[0].startsWith("http")) {
❶
// First argument contains a URI
String serviceURI = args[0];
// Create endpoint reference to service
instanceEPR = new EndpointReferenceType();
instanceEPR.setAddress(new Address(serviceURI));
} else {
❷
```

```
// First argument contains an EPR file name
String eprFile = args[0];
// Get endpoint reference of WS-Resource from file
FileInputStream fis = new FileInputStream(eprFile);
instanceEPR = (EndpointReferenceType) ObjectDeserializer
.deserialize(new InputSource(fis),
EndpointReferenceType.class);
fis.close();
}

❸
// Get PortType
MathPortType math = instanceLocator
.getMathPortTypePort(instanceEPR);

// Perform addition
math.add(value);

// Access value
System.out
.println("Current value: " + math.getValueRP(new GetValueRP()));
} catch (Exception e) {
e.printStackTrace();
}
}
}
```

Note: This file is `$EXAMPLES_DIR/org/globus/examples/clients/MathService_instance/ClientAdd.java`.

❶ If the user specifies a URI, then we create the instance's EPR simply by creating a new `EndpointReferenceType` object and setting it's URI to the one passed as a parameter.

❷ If the user specifies a file, then we create the instance's EPR by reading the XML file using the Globus-supplied `ObjectDeserializer` class.

❸ Finally, we use the instance EPR to obtain a reference to the `MathPortType`. We use this portType to invoke the `add` operation with the value specified in the second parameter of the client.

Now, compile and run the client:

```
javac \
-classpath ./build/stubs/classes/:$CLASSPATH \
org/globus/examples/clients/MathService_instance/ClientAdd.java

java \
-classpath ./build/stubs/classes/:$CLASSPATH \
org.globus.examples.clients.MathService_instance.ClientAdd \
```

```
epr.txt \
10
```

If all goes well, you should see the following:

```
Current value: 10
```

If you run the adder client several times using the same EPR file, you will be able to observe how the value in the resource keeps getting bigger and bigger.

```
Current value: 20
```

```
Current value: 30
```

Chapter 6. Resource Properties

In the previous chapters we have seen how state information in the service is stored inside a resource and, more specifically, in *resource properties*. However, our interaction with resource properties was very limited: our service could modify their values, and we could only access one particular resource property (`Value`) using the `GetValueRP` operation. In this chapter we will see all the some of the tools that will allow us to work with resource properties.

A closer look at resource properties

Before we begin, we need to take a closer look at how resource properties are represented and handled internally in our service. First of all, let's recall how our resource properties are declared in all the examples we've seen so far:

```
<types>
<xsd:schema targetNamespace="http://www.globus.org/namespaces/examples/core/MathService_ins
    xmlns:tns="http://www.globus.org/namespaces/examples/core/MathService_instance"
    xmlns:xsd="http://www.w3.org/2001/XMLSchema">


<!-- REQUESTS AND RESPONSES -->

<!-- ... -->



<!-- RESOURCE PROPERTIES -->

<xsd:element name="Value" type="xsd:int"/>
<xsd:element name="LastOp" type="xsd:string"/>

<xsd:element name="MathResourceProperties">
<xsd:complexType>
<xsd:sequence>
<xsd:element ref="tns:Value" minOccurs="1" maxOccurs="1"/>
<xsd:element ref="tns:LastOp" minOccurs="1" maxOccurs="1"/>
</xsd:sequence>
</xsd:complexType>
</xsd:element>

</xsd:schema>
</types>
```

Notice how we're using XML Schema to declare an element named `MathResourceProperties` that must contain a `Value` element and a `LastOp` element. The `Value` element, in turn, is declared to contain an integer (`xsd:int`) and the `LastOp` element, a string (`xsd:string`).

In the previous examples, we have simply interpreted this as meaning "Our service has two resource properties, `Value` of type integer and `LastOp` of type string". In our Java implementation of the

resource, this simply meant that our resource class had attributes representing each of the resource properties, and that we used special Globus classes (`ReflectionResourceProperty` and `ResourcePropertySet`) to manage those resource properties.

However, the reason why our resource properties are declared in XML Schema, and in that particular way, is because even though they can be implemented internally different ways (not only in GT4, but in other WSRF implementations), they *must* be exchanged with other entities (clients, other services, etc.) as an XML document. This XML representation is called the *resource property document*. For example, the following is an example of how our service's RP document might look like at a given point:

```
<MathResourceProperties xmlns:tns="http://www.globus.org/namespaces/examples/core/MathServi
<tns:Value>50</tns:Value>
<tns:LastOp>ADDITION</tns:LastOp>
</MathResourceProperties>
```

It is important to be familiar with this representation because many operations related with resource properties are better explained in terms of how that operation modifies the RP document. For example, let's suppose our resource properties are declared the following way:

```
<!-- RESOURCE PROPERTIES -->

<xsd:element name="Value" type="xsd:int"/>
<xsd:element name="LastOp" type="xsd:string"/>

<xsd:element name="MathResourceProperties">
<xsd:complexType>
<xsd:sequence>
<xsd:element ref="tns:Value" minOccurs="1" maxOccurs="unbounded"/>
<xsd:element ref="tns:LastOp" minOccurs="1" maxOccurs="unbounded"/>
</xsd:sequence>
</xsd:complexType>
</xsd:element>
```

Notice how we are allowing `Value` and `LastOp` to occur at least one time, with no other limit (`unbounded`). Although internally this will be implemented as an array of integers and an array of strings, the RP document could look like this at a given point:

```
<MathResourceProperties xmlns:tns="http://www.globus.org/namespaces/examples/core/MathServi
<tns:Value>10</tns:Value>
<tns:Value>30</tns:Value>
<tns:Value>50</tns:Value>
<tns:Value>40</tns:Value>
<tns:LastOp>ADDITION</tns:LastOp>
<tns:LastOp>ADDITION</tns:LastOp>
<tns:LastOp>ADDITION</tns:LastOp>
<tns:LastOp>SUBTRACTION</tns:LastOp>
</MathResourceProperties>
```

Later on, for example, we will talk about "inserting a new resource property `LastOp` with value `ADDITION`". This doesn't mean that we are *declaring* a new RP but, rather, that we are inserting a new

element `LastOp` inside our resource property document. Again, it is useful to be aware of how resource properties are represented in XML.

Standard interfaces

An entire WSRF specification, WS-ResourceProperties, is devoted to RPs, RP documents, and to a set of standard portTypes we can use to interact with a service's RPs. Each of these four portTypes exposes a single operation, with the same name as the portType. In this chapter's example we will use all of these portTypes.

GetResourceProperty

This portType allows us to access the value of any resource property given its QName. This portType provides a general way of accessing RPs without the need of an individual `get` operation for each RP (recall that, in previous chapters, we used the `GetValueRP` operation to access the `Value` resource property).

GetMultipleResourceProperties

This portType allows us to access the value of several resource properties at once, given each of their QNames.

SetResourceProperties

This portType allows us to request one or several modifications on a service's RPs. In particular we can perform the following operations:

- Update: Change the value of a RP with a new value.
- Insert: Add a new RP with a given value.
- Delete: Eliminate all occurrences of a certain RP.

Again, note that the `SetResourceProperties` portType has a single operation (not three separate ones). We will use the parameters of the `SetResourceProperties` to specify what action (update, insert, or delete) we want to carry out.

QueryResourceProperties

This portType allows us to perform complex queries on the RP document. Currently, the query language used is XPath.

Accessing resource properties the right way

We will now write and deploy a new service that exposes all the WS-ResourceProperty portTypes. Our client application will, in turn, make a call to some of these portTypes. For simplicity, our service will be based on the example presented in (the example that uses `ServiceResourceHome` to confine our implementation to a single class). However, the steps described in this chapter are equally valid for the other two types of implementations we have seen in the previous two chapters (singleton resource homes, and factory/instance services).

The WSDL file

In the previous chapters, we were always able to reuse our original WSDL file because we were only modifying implementation details (for example, changing the implementation from a singleton resource home to a factory/service approach). However, in this chapter we *do* have to use a new WSDL file because we want to extend from new portTypes, which necessarily changes our service's interface. However, our changes will be minimal:

```
<?xml version="1.0" encoding="UTF-8"?>
<definitions name="MathService"
    targetNamespace="http://www.globus.org/namespaces/examples/core/MathService_instance_rp
    xmlns="http://schemas.xmlsoap.org/wsdl/"
    xmlns:tns="http://www.globus.org/namespaces/examples/core/MathService_instance"
    xmlns:wsdl="http://schemas.xmlsoap.org/wsdl/"
    xmlns:wsrp="http://docs.oasis-open.org/wsrf/2004/06/wsrf-WS-ResourceProperties-1.2-draf
    xmlns:wsrpw="http://docs.oasis-open.org/wsrf/2004/06/wsrf-WS-ResourceProperties-1.2-dra
    xmlns:wsdlpp="http://www.globus.org/namespaces/2004/10/WSDLPreprocessor"
    xmlns:xsd="http://www.w3.org/2001/XMLSchema">

    ❸
<wsdl:import
    namespace=
    "http://docs.oasis-open.org/wsrf/2004/06/wsrf-WS-ResourceProperties-1.2-draft-01.wsdl"
    location="../../wsrf/properties/WS-ResourceProperties.wsdl" />

<-- ... -->

<portType name="MathPortType"
    wsdlpp:extends="wsrpw:GetResourceProperty
                    wsrpw:GetMultipleResourceProperties
                    wsrpw:SetResourceProperties
                    wsrpw:QueryResourceProperties" ❹
    wsrp:ResourceProperties="tns:MathResourceProperties">

<operation name="add">
<input message="tns:AddInputMessage"/>
<output message="tns:AddOutputMessage"/>
</operation>

<operation name="subtract">
<input message="tns:SubtractInputMessage"/>
<output message="tns:SubtractOutputMessage"/>
```

```
</operation>

❺

</portType>

</definitions>
```

Note: This is part of file `$EXAMPLES_DIR/schema/examples/MathService_instance_rp/Math.wsdl`.

❶ Notice how we declare a new target namespace for our new WSDL interface.

❷ This namespace declaration (`wsrpw`) was already present in previous examples. In general, we always need to declare the WS-ResourceProperties namespace if we want to use the portTypes defines in that specification.

❸ Furthermore, we have to make sure we import the WS-ResourceProperties WSDL file, where the portTypes are actually defined.

❹ Our example extends from the four WS-ResourceProperties portTypes. However, we strictly only need to extend from those portTypes we need to use in our service.

❺ Finally, notice how we've eliminated the `GetValueRP` operation (although not shown above, the WSDL file also lacks the corresponding `GetValueRP` messages and elements). Since we are now going to use the `GetResourceProperty` portType, there is no need to expose an explicit `GetValueRP` operation.

Since we have added a new interface, we need to map the new WSDL namespaces to Java packages (as described in)

```
http\://www.globus.org/namespaces/examples/core/MathService_instance_rp=
                                        org.globus.examples.stubs.MathService_instance_rp
http\://www.globus.org/namespaces/examples/core/MathService_instance_rp/bindings=
                                        org.globus.examples.stubs.MathService_instance_rp.bind
http\://www.globus.org/namespaces/examples/core/MathService_instance_rp/service=
                                        org.globus.examples.stubs.MathService_instance_rp.serv
```

Note: These three lines must be present in `$EXAMPLES_DIR/namespace2package.mappings`.

The Java files

The implementation files only require minimal changes. The only noteworthy change is that we no longer need to implement the `getValueRP` operation. In general, using the WS-ResourceProperties portTypes doesn't require that we add any extra code to our Java files.

> **Note:** This QNames interface for this example is `$EXAMPLES_DIR/org/globus/examples/services/core/rp/impl/MathQNames.java`.

> **Note:** This service class for this example is `$EXAMPLES_DIR/org/globus/examples/services/core/rp/impl/MathService.java`.

The deployment files

To be able to use the WS-ResourceProperties portTypes we need to modify our WSDD file to make sure that our service relies on the Globus-supplied operation providers for those portTypes.

```
<?xml version="1.0" encoding="UTF-8"?>
<deployment name="defaultServerConfig"
    xmlns="http://xml.apache.org/axis/wsdd/"
    xmlns:java="http://xml.apache.org/axis/wsdd/providers/java"
    xmlns:xsd="http://www.w3.org/2001/XMLSchema">

    <service name="examples/core/rp/MathService" provider="Handler" use="literal" style="do
        <parameter name="className" value="org.globus.examples.services.core.rp.impl.MathSe
        <wsdlFile>share/schema/examples/MathService_instance_rp/Math_service.wsdl</wsdlFile
        <parameter name="allowedMethods" value="*"/>
        <parameter name="handlerClass" value="org.globus.axis.providers.RPCProvider"/>
        <parameter name="scope" value="Application"/>
        <parameter name="providers" value="GetRPProvider GetMRPProvider SetRPProvider Query
        <parameter name="loadOnStartup" value="true"/>
    </service>

</deployment>
```

> **Note:** This is file `$EXAMPLES_DIR/org/globus/examples/services/core/rp/deploy-server.wsdd`.

> **Operation Providers:** GT4 Core uses a design pattern called *operation providers* that will make our lives as programmers much easier. To put it quite simply, an operation provider is a Java class, providing a set of operations, that we can easily plug into our service. For example, remember that the WSDL file we've used in the previous chapters included the following:
>
> ```
> <portType name="MathPortType"
> ```

```
wsdlpp:extends="wsrpw:GetResourceProperty"
wsrp:ResourceProperties="tns:MathResourceProperties">
```

We used the `wsdlpp:extends` attribute to specify that our service would also implement a standard WSRF portType: the GetResourceProperty portType. This means our service class would need to implement a GetResourceProperty method (which we'll see in more detail in the next chapter). However, instead of having to implement it ourselves, we can rely on the operation providers included with GT4 that *provide* an implementation of all the WSRF portTypes. To specify we wanted to use an operation provider in our service, we simply added the following to our WSDD file:

```
<parameter name="providers" value="GetRPProvider"/>
```

In the following chapters, each time we want our service to provide standard functionality specified in the WSRF specs, we will simply make our service extend from a standard WSRF portType and then 'plug in' a Globus operation provider that implements that portType.

The JNDI deployment file, on the other hand, doesn't require any changes.

Build and deploy

Build the service:

```
./globus-build-service.sh rp
```

And deploy it:

```
globus-deploy-gar $EXAMPLES_DIR/org_globus_examples_services_core_rp.gar
```

Client code

Our client application will make calls to some of the WS-ResourceProperties portTypes. The next example will use more complex resource properties and then we will see how to invoke the rest of the portTypes.

The code for this client is rather lengthy, so instead of seeing all the code all at once, we are going to run the client first, and then take a close look at what happens at each moment.

Note: This source code for the client is `$EXAMPLES_DIR/org/globus/examples/clients/MathService_instance_rp/Client.java`

Compile the client:

```
javac \
-classpath ./build/stubs/classes/:$CLASSPATH \
org/globus/examples/clients/MathService_instance_rp/Client.java
```

And run it:

```
java \
-classpath ./build/stubs/classes/:$CLASSPATH \
org.globus.examples.clients.MathService_instance_rp.Client \
http://127.0.0.1:8080/wsrf/services/examples/core/rp/MathService
```

The full output of the client should be the following:

```
Value RP: 0
LastOp RP: NONE
Value RP: 10
LastOp RP: ADDITION

Value RP: 100
LastOp RP: ADDITION

Value: 100
LastOp: ADDITION
```

Let's take a close look at what happens in each of these three blocks.

Invoking `getResourceProperty`

```
Value RP: 0
LastOp RP: NONE
Value RP: 10
LastOp RP: ADDITION
```

The first block of code prints out the initial values of the `Value` and `LastOp` RP's using the `getResourceProperty` operation, performs an addition, and then prints out the RP's again. All the `getResourceProperty` code is placed inside a `printResourceProperties` method.

```
printResourceProperties(math);
math.add(10);
printResourceProperties(math);
```

Let's take a close look at what happens in the `printResourceProperties` method:

```
/*
 * This method prints out MathService's resource properties by using the
 * GetResourceProperty operation.
 */
private void printResourceProperties(MathPortType math) throws Exception {
GetResourcePropertyResponse valueRP, lastOpRP, lastLogRP;
String value, lastOp, lastLog;

❶
valueRP = math.getResourceProperty(MathQNames.RP_VALUE);
lastOpRP = math.getResourceProperty(MathQNames.RP_LASTOP);

❷
value = valueRP.get_any()[0].getValue();
lastOp = lastOpRP.get_any()[0].getValue();
```

```
❸
System.out.println("Value RP: " + value);
System.out.println("LastOp RP: " + lastOp);
}
```

❶ We first invoke the `getResourceProperty` operation on our portType. Take into account that, since our `MathPortType` portType *extends* from the standard `GetResourceProperty` portType, our portType also includes a `getResourceProperty` operation. The only parameter we have to include is the QName of the RP we want to retrieve. Notice how the return value is of type `GetResourcePropertyResponse`, a Globus-supplied stub class.

❷ We must now extract the actual value of the RP's from the `GetResourcePropertyResponse` return value. This is when knowing about the *resource property document* (explained at the beginning of the chapter) comes in really handy. The `GetResourcePropertyResponse` object will contain zero, one, or many RPs in XML format (i.e. the same way they are represented in the RP document). To access these RPs, we need to use the `get_any` method, which returns an array of *elements* (in the XML sense of the word). In our case, the `GetResourcePropertyResponse` from requesting the `Value` RP will contain the following:

```
<ns1:Value xmlns:ns1="http://www.globus.org/namespaces/examples/core/MathService_instanc
```

To obtain the value `0` contained in that element, we simply need to access the first position of the array of elements (`get_value()[0]`) and get its value (`getValue`).

❸ Finally, we print out the values.

Invoking `SetResourceProperties` to update

```
Value RP: 100
LastOp RP: ADDITION
```

The second block of code updates the value of the `Value` RP using the `SetResourceProperties` operation and requesting an `Update` action. All the update code is placed inside a `updateRP` method.

```
updateRP(endpoint, MathQNames.RP_VALUE, "100");
printResourceProperties(math);
```

Now, let's see how the update operation is actually carried out:

```
/*
 * This method updates resource property "rpQName" in the WS-Resource
 * pointed at by the endpoint reference "epr" with the new value "value".
 */
private void updateRP(EndpointReferenceType epr, QName rpQName, String value)
throws Exception {
❶
WSResourcePropertiesServiceAddressingLocator locator = new WSResourcePropertiesServiceAddre
SetResourceProperties_PortType port = locator
```

```
.getSetResourcePropertiesPort(epr);

❷
UpdateType update = new UpdateType();
MessageElement msg = new MessageElement(rpQName, value);
update.set_any(new MessageElement[] { msg });

❸
SetResourceProperties_Element request = new SetResourceProperties_Element();
request.setUpdate(update);

❹
port.setResourceProperties(request);
}
```

❶ First of all, we obtain a reference to a generic `SetResourceProperties` portType. This approach is different from the one used in the previous block of code, where we simply used our own `MathPortType`. Take into account that we *could* use our `MathPortType` to invoke the `setResourceProperties` operation. However, the approach followed here can come in handy when all we want to access is the standard WSRF operations, without having to get a reference to the full portType (in our case, `MathPortType`).

❷ Since we are going to perform an update action through the `SetResourceProperties` operation, we first need to create an `UpdateType` object where we specify the update to carry out. Take into account that an `UpdateType` object can contain *several* update requests. We encapsulate each of these requests inside a `MessageElement` object. Then, we create an array of `MessageElements` and include that array in our `UpdateType` object (using the `set_any` method).

❸ Now, we create a `SetResourceProperties_Element` object which will represent our `SetResourceProperties` request. This object can contain insert, update, and delete actions. In our case, we add the recently created `UpdateType` object to the request using the `setUpdate` method.

❹ Finally, we invoke `SetResourceProperties`.

Invoking `GetMultipleResourceProperties`

```
Value: 100
LastOp: ADDITION
```

The third, and last, block of code prints out the values of the `Value` and `LastOp` RP's using the `GetMultipleResourceProperties` operation. All the `GetMultipleResourceProperties` code is placed inside a `printMultipleResourceProperties` method.

```
printMultipleResourceProperties(math);


/*
 * This method prints out MathService's resource properties by using the
 * GetMultipleResourceProperties operation.
```

```
 */
private void printMultipleResourceProperties(MathPortType math)
throws Exception {
GetMultipleResourceProperties_Element request;
GetMultipleResourcePropertiesResponse response;

❶
QName[] resourceProperties = new QName[] { MathQNames.RP_VALUE,
MathQNames.RP_LASTOP };
request = new GetMultipleResourceProperties_Element(resourceProperties);

❷
response = math.getMultipleResourceProperties(request);

❸
for(int i=0; i<response.get_any().length;i++)
{
String name = response.get_any()[i].getLocalName();
String value = response.get_any()[i].getValue();
System.out.println(name +": " + value);
}
}
```

❶ First, we need to create a `GetMultipleResourceProperties_Element` object that represents the request to `getMultipleResourceProperties`. The constructor expects an array of QNames. In our case, we specify the QNames for the `Value` and `LastOp` RPs

❷ Next, we invoke the `getMultipleResourceProperties`. Notice how the return value is of type `GetMultipleResourcePropertiesResponse`.

❸ As in `getResourceProperty`, the return of `getMultipleResourceProperties` encapsulates zero, one, or many RPs in XML format. In this case, the `GetMultipleResourcePropertiesResponse` will contain the following:

```
<ns1:Value xmlns:ns1="http://www.globus.org/namespaces/examples/core/MathService_instanc
<ns2:LastOp xmlns:ns2="http://www.globus.org/namespaces/examples/core/MathService_instan
```

To extract the value of the RPs, we once again rely on the `get_any` method, which returns an array of elements. We simply have to iterate through this array, and write the value of each element using the `getValue` method. Here we are also printing out the name of the property using the `getLocalName` method.

Chapter 7. Lifecycle Management

In this chapter we will see the two lifecycle management solutions offered by the WS-ResourceLifetime specification. Since lifecycle management mainly makes sense when we have several resources, the examples will focus on explaining what modifications are necessary to the example seen in (the factory/instance example). The version of that example with the lifecycle modifications can be found in directory `$EXAMPLES_DIR/org/globus/examples/services/core/rl/`

Immediate destruction

Immediate destruction is the simplest type of lifecycle management. It allows us to request that a resource be destroyed immediately by invoking a `destroy` operation in the *instance* service. Notice how, even though the *factory* service is responsible for creating the resources, destruction must be requested to each individual resource through the instance service.

To add immediate destruction to our service, we simply need to extend from the standard WSRF `ImmediateResourceTermination` portType. This portType adds a `destroy` operation to our portType that will instruct the current resource to terminate itself immediately.

```
<portType name="MathPortType"
    wsdlpp:extends="wsrpw:GetResourceProperty
                    wsrlw:ImmediateResourceTermination"
    wsrp:ResourceProperties="tns:MathResourceProperties">

<operation name="add">
<input message="tns:AddInputMessage"/>
<output message="tns:AddOutputMessage"/>
</operation>

<operation name="subtract">
<input message="tns:SubtractInputMessage"/>
<output message="tns:SubtractOutputMessage"/>
</operation>

</portType>
```

To be able to do this, we must remember to declare the WS-ResourceLifetime namespace, and import its WSDL file:

```
<definitions name="MathService"
    targetNamespace="http://www.globus.org/namespaces/examples/core/MathService_instance_rl
    xmlns="http://schemas.xmlsoap.org/wsdl/"
    xmlns:tns="http://www.globus.org/namespaces/examples/core/MathService_instance_rl"
    xmlns:wsdl="http://schemas.xmlsoap.org/wsdl/"
    xmlns:wsrlw="http://docs.oasis-open.org/wsrf/2004/06/wsrf-WS-ResourceLifetime-1.2-draft
    xmlns:wsrp="http://docs.oasis-open.org/wsrf/2004/06/wsrf-WS-ResourceProperties-1.2-draf
    xmlns:wsrpw="http://docs.oasis-open.org/wsrf/2004/06/wsrf-WS-ResourceProperties-1.2-dra
    xmlns:wsdlpp="http://www.globus.org/namespaces/2004/10/WSDLPreprocessor"
    xmlns:xsd="http://www.w3.org/2001/XMLSchema">
```

```
<wsdl:import
    namespace=
    "http://docs.oasis-open.org/wsrf/2004/06/wsrf-WS-ResourceLifetime-1.2-draft-01.wsdl"
    location="../../wsrf/lifetime/WS-ResourceLifetime.wsdl" />
```

Note: This is part of file `$EXAMPLES_DIR/schema/examples/MathService_instance_rl/Math.wsdl`

Next, we need to add the Globus-supplied `DestroyProvider` operation provider to the instance service. This provider implements the `destroy` operation mentioned above.

```
<?xml version="1.0" encoding="UTF-8"?>
<deployment name="defaultServerConfig"
    xmlns="http://xml.apache.org/axis/wsdd/"
    xmlns:java="http://xml.apache.org/axis/wsdd/providers/java"
    xmlns:xsd="http://www.w3.org/2001/XMLSchema">

    <!-- Instance service -->
    <service name="examples/core/rl/MathService" provider="Handler" use="literal" style="do
        <parameter name="className" value="org.globus.examples.services.core.rl.impl.MathSe
        <wsdlFile>share/schema/examples/MathService_instance_rl/Math_service.wsdl</wsdlFile
        <parameter name="allowedMethods" value="*"/>
        <parameter name="handlerClass" value="org.globus.axis.providers.RPCProvider"/>
        <parameter name="scope" value="Application"/>
        <parameter name="providers" value="GetRPProvider DestroyProvider"/>
    </service>

    <!-- Factory service -->
    <service name="examples/core/rl/MathFactoryService" provider="Handler" use="literal" st
        <parameter name="className" value="org.globus.examples.services.core.rl.impl.MathFa
        <wsdlFile>share/schema/examples/FactoryService/Factory_service.wsdl</wsdlFile>
        <parameter name="allowedMethods" value="*"/>
        <parameter name="handlerClass" value="org.globus.axis.providers.RPCProvider"/>
        <parameter name="scope" value="Application"/>
        <parameter name="instance" value="examples/core/rl/MathService"/>
    </service>

</deployment>
```

Note: This file is `$EXAMPLES_DIR/org/globus/examples/services/core/rl/deploy-server.wsdd`.

Now, we can compile the service:

```
./globus-build-service.sh rl
```

And deploy it:

```
globus-deploy-gar $EXAMPLES_DIR/org_globus_examples_services_core_rl.gar
```

To try out resource destruction, we will use a client that is identical to the simple client seen in . The only difference is that, at the end of the client we will add a call to the `destroy` operation.

```
package org.globus.examples.clients.FactoryService_Math_rl;

import org.apache.axis.message.addressing.Address;
import org.apache.axis.message.addressing.EndpointReferenceType;

import org.globus.examples.stubs.MathService_instance_rl.MathPortType;
import org.globus.examples.stubs.MathService_instance_rl.service.MathServiceAddressingLocat
import org.globus.examples.stubs.Factory.service.FactoryServiceAddressingLocator;
import org.globus.examples.stubs.Factory.FactoryPortType;
import org.globus.examples.stubs.Factory.CreateResource;
import org.globus.examples.stubs.Factory.CreateResourceResponse;

import org.oasis.wsrf.lifetime.Destroy;

/* This client creates a new MathService instance through a FactoryService. This client
 * expects one parameter: the factory URI.
 */
public class Client_immed {

public static void main(String[] args) {
FactoryServiceAddressingLocator factoryLocator = new FactoryServiceAddressingLocator();
MathServiceAddressingLocator instanceLocator = new MathServiceAddressingLocator();

try {
String factoryURI = args[0];
EndpointReferenceType factoryEPR, instanceEPR;
FactoryPortType mathFactory;
MathPortType math;

// Get factory portType
factoryEPR = new EndpointReferenceType();
factoryEPR.setAddress(new Address(factoryURI));
mathFactory = factoryLocator.getFactoryPortTypePort(factoryEPR);

// Create resource and get endpoint reference of WS-Resource.
// This resource is our "instance".
CreateResourceResponse createResponse = mathFactory
.createResource(new CreateResource());
instanceEPR = createResponse.getEndpointReference();

// Get instance PortType
math = instanceLocator.getMathPortTypePort(instanceEPR);

System.out.println("Created instance.");

// Perform an addition
math.add(10);
```

```
// Perform another addition
math.add(5);

// Perform a subtraction
math.subtract(5);

math.destroy(new Destroy());
System.out.println("Destroyed instance.");
} catch (Exception e) {
e.printStackTrace();
}
}

}
```

> **Note:** This file is `$EXAMPLES_DIR/org/globus/examples/clients/FactoryService_Math_rl/Client_immed.java`

Compile the client:

```
javac \
-classpath ./build/stubs/classes/:$CLASSPATH \
org/globus/examples/clients/FactoryService_Math_rl/Client_immed.java
```

And run it:

```
java \
-classpath ./build/stubs/classes/:$CLASSPATH \
org.globus.examples.clients.FactoryService_Math_rl.Client_immed \
http://127.0.0.1:8080/wsrf/services/examples/core/rl/MathFactoryService
```

If all goes well, you should see the following:

```
Created instance.
Destroyed instance.
```

Well, that wasn't too exciting, was it? How do we really now that resource destruction is actually happening? Well, there's a simple way of testing it. Modify the last lines of the client so they will look like so:

```
math.destroy(new Destroy());
System.out.println("Destroyed instance.");

// Perform another addition
math.add(5);
```

As you can see, we are going to try to invoke an operation *after* destroying the resource that operation is supposed to use. As you can probably imagine, no good will come of this. If you recompile the client and run it again, you should again see the following:

```
Created instance.
```

```
Destroyed instance.
```

And, then, a really nasty error message where you should be able to make out the following:

```
java.rmi.RemoteException: ; nested exception is:
        org.globus.wsrf.NoSuchResourceException
```

What has just happened is that the `add` has been invoked as normal. However, the endpoint reference that is being passed in the call refers to a resource that no longer exists. So, when `add` tries to retrieve the resource, a `NoSuchResourceException` is thrown.

Scheduled destruction

Scheduled destruction is a more elaborate form of resource lifecycle management, as it allows us to specify exactly when we want the resource to be destroyed. The main application of scheduled destruction is to perform *lease-based lifecycle management*, where we initially set the destruction time of a resource some time in the future (for example, 5 minutes). This is called the *lease*. Our application must periodically *renew the lease* (setting the destruction time another 5 minutes in the future), or the resource will eventually be destroyed. This will allow our application to purge resources that for some reason (network failure, programmer errors, etc.) have become unavailable (and therefore can't receive the lease renewal).

Using scheduled destruction requires adding more code that immediate destruction because the standard WSRF portType that provides scheduled destruction not only adds a new operation (`SetTerminationTime`) but also two new resource properties: `TerminationTime` and `CurrentTime`. `TerminationTime` specifies when the resource is set to be destroyed, and the value of `CurrentTime` must always be the time in the machine that hosts the resource. This means that, not only will we have to modify the WSDL file, we will also have to make sure those two new resource properties are properly implemented in our resource class.

The WSDL file

So, let's start with the easy part. To use scheduled resource termination, our portType must extend from the `ScheduledResourceTermination` portType:

```
<portType name="MathPortType"
    wsdlpp:extends="wsrpw:GetResourceProperty
                    wsrlw:ScheduledResourceTermination"
    wsrp:ResourceProperties="tns:MathResourceProperties">

<operation name="add">
<input message="tns:AddInputMessage"/>
<output message="tns:AddOutputMessage"/>
</operation>

<operation name="subtract">
<input message="tns:SubtractInputMessage"/>
<output message="tns:SubtractOutputMessage"/>
```

```
</operation>

</portType>
```

> **Note:** As seen in immediate destruction, we mustn't forget to declare the WS-ResourceLifetime namespace (`wsrlw`), and import its WSDL file.

The resource implementation

Next, we have to implement the `ResourceLifetime` interface in our resource class. This interface requires that we provide get/set methods for the `TerminationTime` and `CurrentTime` RPs.

```
public class MathResource implements Resource, ResourceIdentifier,
ResourceProperties, ResourceLifetime
```

So, we'll start by adding a `terminationTime` attribute of type `Calendar` to our class to represent the resource's termination time. We don't need to add a `currentTime` attribute because that RP's get method will always return the system's time (which, as we'll see, we can easily obtain using the Java API).

```
/* Resource properties */
private int value;
private String lastOp;
private Calendar terminationTime;
```

Now, we have to make sure we add the two RPs to our resource's RP set:

```
/* Initializes RPs and returns a unique identifier for this resource */
public Object initialize() throws Exception {
this.key = new Integer(hashCode());
this.propSet = new SimpleResourcePropertySet(
MathQNames.RESOURCE_PROPERTIES);

try {
ResourceProperty valueRP = new ReflectionResourceProperty(
MathQNames.RP_VALUE, "Value", this);
this.propSet.add(valueRP);
setValue(0);

ResourceProperty lastOpRP = new ReflectionResourceProperty(
MathQNames.RP_LASTOP, "LastOp", this);
this.propSet.add(lastOpRP);
setLastOp("NONE");

ResourceProperty termTimeRP = new ReflectionResourceProperty(
SimpleResourcePropertyMetaData.TERMINATION_TIME, this);
this.propSet.add(termTimeRP);
```

```
ResourceProperty currTimeRP = new ReflectionResourceProperty(
SimpleResourcePropertyMetaData.CURRENT_TIME, this);
this.propSet.add(currTimeRP);

} catch (Exception e) {
throw new RuntimeException(e.getMessage());
}

return key;
}
```

Notice we use a Globus-supplied `SimpleResourcePropertyMetaData` class which includes information on the `TerminationTime` and `CurrentTime` RPs. We must make sure we import this class:

```
import org.globus.wsrf.impl.SimpleResourcePropertyMetaData;
```

Finally, the last thing needed in the resource implementation is to add a `get` and `set` method for the `TerminationTime` RP, and a `get` method for the `CurrentTime` RP (we can't 'set' the current time). Notice how we return the current time using an instance of the Java `Calendar` class.

```
/* Required by interface ResourceLifetime */
public Calendar getCurrentTime() {
return Calendar.getInstance();
}

public Calendar getTerminationTime() {
return this.terminationTime;
}

public void setTerminationTime(Calendar terminationTime) {
this.terminationTime=terminationTime;
}
```

> **Note:** This is part of file `$EXAMPLES_DIR/org/globus/examples/services/core/rl/impl/MathResource.java`

Deployment

As we did in immediate destruction, we need to add a Globus-supplied operation provider, `SetTerminationTimeProvider`, to the instance service. This provider implements the `setTerminationTime` operation that will allow us to set the resource's termination time. Note that we *cannot* set the termination time by directly modifying the `TerminationTime` RP (using, for example, the `SetResourceProperties` operation). We must use the `setTerminationTime` operation (this operation, as implemented in the Globus-supplied operation provider, does more than just update the RP).

```
<parameter name="providers" value="GetRPProvider SetTerminationTimeProvider"/>
```

> **Note:** This is part of file `$EXAMPLES_DIR/org/globus/examples/services/core/rl/deploy-server.wsdd`.

We can also modify the JNDI deploy file to control how often the container will check if a resource is past its termination time. This is done with the `sweeperDelay` parameter, specified in milliseconds. The default value is to check every one minute (60000 milliseconds). We will change this value to one second (1000 milliseconds) so our client will be able to observe how the resource does, in fact, expire.

```
<parameter>
<name>sweeperDelay</name>
<value>1000</value>
</parameter>
```

> **Note:** This is part of file `$EXAMPLES_DIR/org/globus/examples/services/core/rl/deploy-jndi-config.xml`.

Finally, build and deploy:

```
./globus-build-service.sh rl
```

```
globus-deploy-gar $EXAMPLES_DIR/org_globus_examples_services_core_rl.gar
```

The client

We will test our service by creating a new resource, setting its termination 10 seconds in the future, and then checking every second to see if the resource is still 'alive'. When the resource is terminated, any call to the resource will produce an exception. Like the immediate destruction client, this client is similar to the simple client seen in . The following is the code that we will run after the resource has been created:

❶
```
Calendar termination = Calendar.getInstance();
termination.add(Calendar.SECOND, 10);
```

❷
```
SetTerminationTime request;
SetTerminationTimeResponse response;
request = new SetTerminationTime(termination);
response = math.setTerminationTime(request);
```

❸
```
System.out.println("Current time               "
+ response.getCurrentTime().getTime());
System.out.println("Requested termination time "
+ termination.getTime());
System.out.println("Scheduled termination time "
+ response.getNewTerminationTime().getTime());
```

```
boolean terminated = false;
int seconds = 0;
while (!terminated) { ❹
try {
System.out.println("Second " + seconds);
math.add(10);
Thread.sleep(1000);
seconds++;
} catch (RemoteException e) {
System.out.println("Resource has been destroyed");
terminated = true;
}
}
```

> **Note:** This is part of file
> `$EXAMPLES_DIR/org/globus/examples/clients/FactoryService_Math_rl/Client_sched.java`

❶ We get an instance of the `Calendar` class, which contains the current time. We add 10 seconds to it. This will be the termination time of our resource.

❷ We make a call to the `SetTerminationTime` operation, sending the new termination time.

❸ The response from the `SetTerminationTime` operation returns interesting information: the resource's current time and the *scheduled* termination, which might differ from the requested termination time (in simple scenarios like the one we are trying out now, this will not happen).

❹ Finally, this loop makes a call to the `add` operation every second. When the resource is finally destroyed, and an exception is thrown, we exit the loop.

Compile and run the client:

```
javac \
-classpath ./build/stubs/classes/:$CLASSPATH \
org/globus/examples/clients/FactoryService_Math_rl/Client_sched.java
```

```
java \
-classpath ./build/stubs/classes/:$CLASSPATH \
org.globus.examples.clients.FactoryService_Math_rl.Client_sched \
http://127.0.0.1:8080/wsrf/services/examples/core/rl/MathFactoryService
```

If all goes well, you should see the following:

```
Created instance.
Current time               Sun Apr 03 00:54:29 CST 2005
Requested termination time Sun Apr 03 00:54:39 CST 2005
Scheduled termination time Sun Apr 03 00:54:39 CST 2005
Second 0
Second 1
```

```
Second 2
Second 3
Second 4
Second 5
Second 6
Second 7
Second 8
Second 9
Second 10
Resource has been destroyed
```

Chapter 8. Notifications

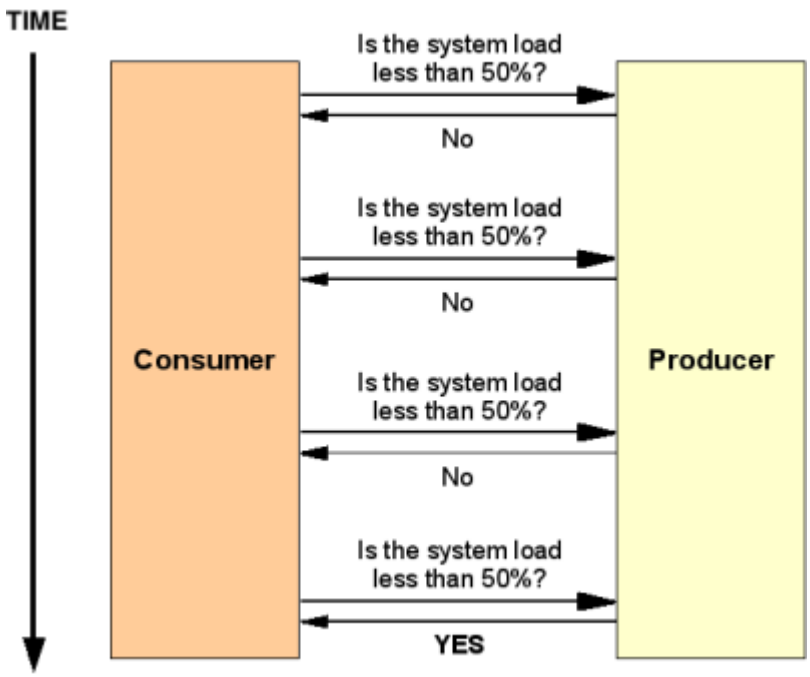
In this chapter we will introduce the concepts of `notification`, a common design pattern that allows clients to be notified when interesting events happen in a server. In particular, we will focus on WS-Notifications, a family of specification that allow us to use this design pattern with Web Services. Then, we will see two examples of how we can use notifications in our services.

What are notifications?

Notifications are nothing new. It's a very popular software design pattern, although you might know it with a different name such as Observer/Observable. Let's suppose that our software had several distinct parts (e.g. a GUI and the application logic, a client and a server, etc.) and that one of the parts of the software needs to be aware of the changes that happen in one of the other parts. For example, the GUI might need to know when a value is changed in a database, so that the new value is immediately displayed to the user. Taking this to the client/server world is easy: suppose a client needs to know when the server reaches a certain state, so the client can perform a certain action.

The most crude approach to keep the client informed is a *polling* approach. The client periodically *polls* the server (asks if there are any changes). For example, let's suppose a client applications wants to know when the load of a server drops below 50%. The server is called the *producer* of events (in this case, the event is a drop in the server load). The client, on the other hand, is called the *consumer* of events. The polling approach would go like this:

Figure 8-1. Keeping track of changes using polling



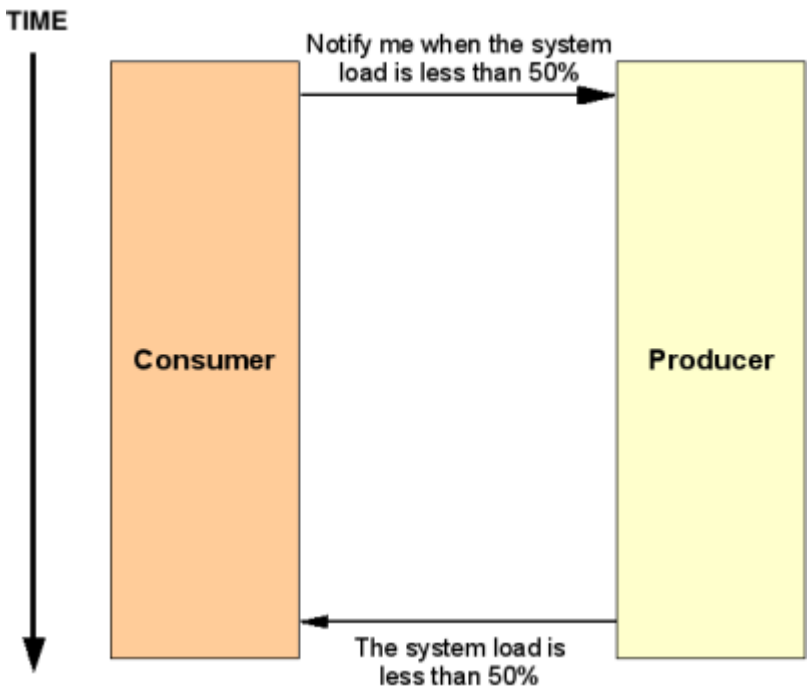
1. The consumer asks the producer if there are any changes. The producer replies "No", so the consumer waits a while before making another call.
2. Once again, the consumer asks the producer if there are any changes. The producer replies "No", so the consumer waits a while before making another call.
3. As you can see, this step can be repeated *ad nauseam* until the server finally replies that there has been a change.

This approach isn't very efficient, specially if you consider the following:

- If the time between calls is very small, the amount of network traffic and CPU use increases.
- There can be more than one consumer. If we have dozens of consumers, waiting for an event to happen, then the producer could get saturated with calls asking it if there are any changes.

The answer to this problem is actually terribly simple (and common sense). Instead of periodically asking the producer if there are any changes, we make an initial call asking the producer to *notify* the consumer whenever a certain event occurs. This is the *notification* approach.

Figure 8-2. Keeping track of changes using notifications



1. The consumer asks the producer to notify him as soon as the server load drops below 50%. The producer keeps a list of all its registered consumers. This step is normally called the *subscription* or *registration* step.
2. The consumer and the producer go about their business until the server load drops below 50%.

3. Once the server load drops below 50%, the producer *notifies* all its consumers (remember, there can be more than one) of that event.

As you can see, this approach is much more efficient (in this simple example, network traffic has been sliced in half with respect to the polling approach).

WS-Notifications

The WS-Notifications family of specifications, although not a part of WSRF, has strong ties to it. It provides a set of standard interfaces to use the notification design pattern with Web Services. WS-Notifications is divided into three specifications: WS-Topics, WS-BaseNotification, and WS-BrokeredNotification.

WS-Topics

First of all, we have *topics*, which are used by the other two specifications in WS-Notifications to present a set of "items of interest for subscription". As we will see next, a service can publish a set of topics that clients can subscribe to, and receive a notification whenever the topic changes. Topics are very versatile, as they even allow us to create *topic trees*, where a topic can have a set of *child topics*. By subscribing to a topic, a client automatically receives notifications from all the descendant topics (without having to manually subscribe to each of them).

WS-BaseNotification

This specification defines the standard interfaces of notification consumers and producers. In a nutshell, notification producers have to expose a *subscribe* operation that notification consumers can use to request a subscription. Consumers, in turn, have to expose a *notify* operation that producers can use to deliver the notification. Furthermore, the client actually requesting the subscription need not necessarily be the consumer of those notifications. In other words, clients can perform subscriptions "on behalf of other notification consumers".

Figure 8-3. A typical WS-Notification interaction

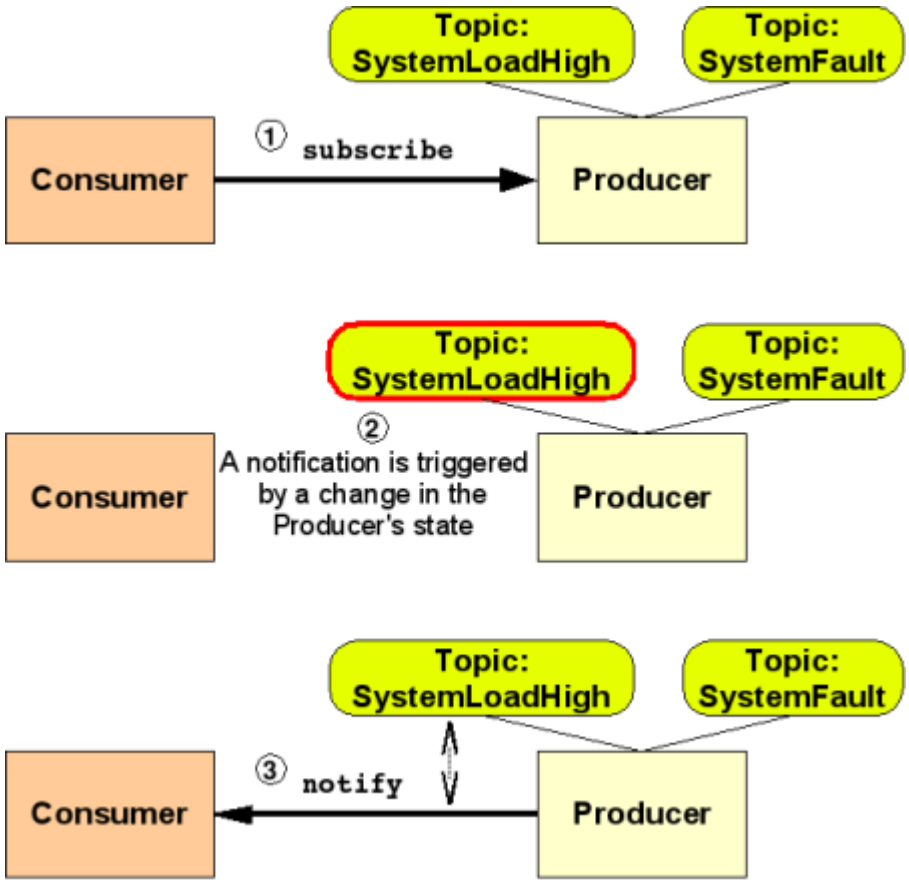


Figure 8-3 shows an example interaction between a notification consumer and producer, in the simple case when the subscriber and consumer are the same entity. In this example we have a single notification consumer, and a single notification producer that publishes two topics: `SystemLoadHigh` and `SystemFault`.

1. First of all, the notification consumer subscribes himself to the `SystemLoadHigh` topic. It is interesting to note that, internally, a `Subscription` resource is created with information regarding the subscription (not shown in the figure).
2. Next, at some point in time, something happens in the notification producer that must trigger a notification from the `SystemLoadHigh` topic. For example, we might have implemented our service to send out a notification every time the system load passes from "more than 50%" to "less than 50%".
3. The notification producer delivers the notification to the consumer by invoking the `notify` operation in the consumer. As shown in the figure, this notification delivery is tied to the topic that triggered the notification.

Figure 8-4. A WS-Notification interaction where the subscriber and the consumer are different

entities

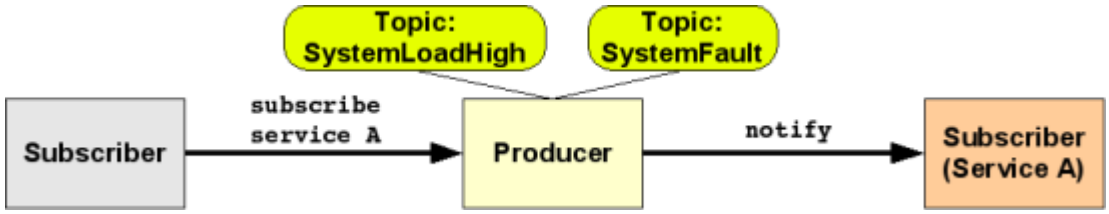


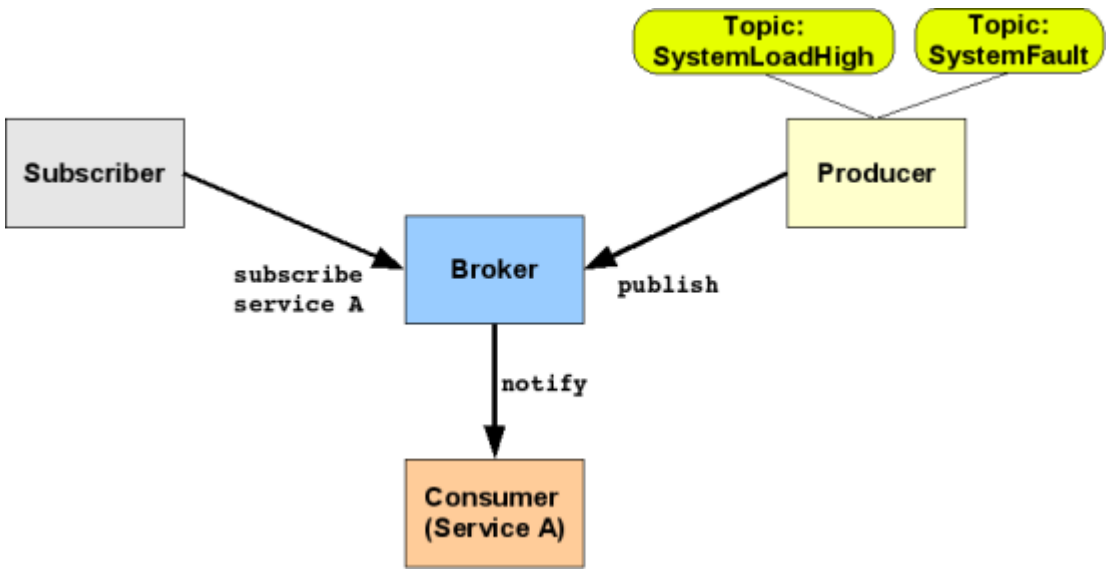
Figure 8-4, on the other hand, shows how the subscriber and the consumer need not be the same entity. In the figure, the subscriber requests the producer that Service A be subscribed to the `SystemLoadHigh` topic. When a notification is triggered, the notification is sent to Service A (the consumer) not to the subscriber.

WS-BrokeredNotification

In brokered notifications we consider the case when notifications are delivered from the producer to the consumer through an intermediate entity called the *broker*. The WS-BrokeredNotification defines the standard interfaces for the notification broker.

As shown in Figure 8-5, in the presence of a notification broker, the producer must register with the broker and publish its topics there. The subscriber (separate from the consumer in this case), must also subscribe through the broker, not directly with the producer. Finally, when a notification is produced, it is delivered to the consumer through the broker.

Figure 8-5. A typical brokered WS-Notification interaction



Notifications in GT4

GT4 currently doesn't implement the WS-Notifications family of specifications completely. For example, no support for brokered notification is included. However, GT4 does allow us to perform effective topic-based notification. One of the more interesting parts of the GT4 implementation of WS-Notifications is that it will allow us to effortlessly expose a resource property as a topic, triggering a notification each time the value of the RP changes. We will also be able to define our own topics, which need not trigger a notification every single time the value of an RP changes. In the remainder of the chapter, we will see how we can add both types of topics to our service.

Notifying changes in a resource property

We will see how we can add notifications to a service so clients can be notified each time a certain RP is modified. As we did in and , our example will be based, for simplicity, on the `ServiceResourceHome` resource home.

The WSDL file

Our portType will need to extend from a standard WS-Notifications portType called `NotificationProducer`, which exposes a `Subscribe` operation that consumers can use to subscribe themselves to a particular topic. Since this examples takes an existing resource property and exposes it as a topic, no additional WSDL code is required beyond extending the `NotificationProducer` portType.

First of all, we need to declare the WS-Notifications namespace, and import its WSDL file.

```
<?xml version="1.0" encoding="UTF-8"?>
<definitions name="MathService"
    targetNamespace="http://www.globus.org/namespaces/examples/core/MathService_instance_no
    xmlns="http://schemas.xmlsoap.org/wsdl/"
    xmlns:tns="http://www.globus.org/namespaces/examples/core/MathService_instance_notif"
    xmlns:wsdl="http://schemas.xmlsoap.org/wsdl/"
    xmlns:wsrp="http://docs.oasis-open.org/wsrf/2004/06/wsrf-WS-ResourceProperties-1.2-draf
    xmlns:wsrpw="http://docs.oasis-open.org/wsrf/2004/06/wsrf-WS-ResourceProperties-1.2-dra
    xmlns:wsntw="http://docs.oasis-open.org/wsn/2004/06/wsn-WS-BaseNotification-1.2-draft-0
    xmlns:wsdlpp="http://www.globus.org/namespaces/2004/10/WSDLPreprocessor"
    xmlns:xsd="http://www.w3.org/2001/XMLSchema">

<wsdl:import
    namespace=
    "http://docs.oasis-open.org/wsrf/2004/06/wsrf-WS-ResourceProperties-1.2-draft-01.wsdl"
    location="../../wsrf/properties/WS-ResourceProperties.wsdl" />

<wsdl:import
    namespace=
    "http://docs.oasis-open.org/wsn/2004/06/wsn-WS-BaseNotification-1.2-draft-01.wsdl"
    location="../../wsrf/notification/WS-BaseN.wsdl"/>


    <!-- ... -->
```

```
</definitions>
```

Then, we need to extend from the `NotificationProducer` portType.

```
<portType name="MathPortType"
    wsdlpp:extends="wsrpw:GetResourceProperty wsntw:NotificationProducer"
    wsrp:ResourceProperties="tns:MathResourceProperties">

  <!-- <operation>s -->
</portType>
```

Note: This is part of file `$EXAMPLES_DIR/schema/examples/MathService_instance_notif/Math.wsdl`.

Finally, since these modifications create a new interface, we need to map the new WSDL namespaces to Java packages.

```
http\://www.globus.org/namespaces/examples/core/MathService_instance_notif=
                                    org.globus.examples.stubs.MathService_instance_notif
http\://www.globus.org/namespaces/examples/core/MathService_instance_notif/bindings=
                                    org.globus.examples.stubs.MathService_instance_notif.b
http\://www.globus.org/namespaces/examples/core/MathService_instance_notif/service=
                                    org.globus.examples.stubs.MathService_instance_notif.s
```

Note: These three lines must be present in `$EXAMPLES_DIR/namespace2package.mappings`.

The resource implementation

`SimpleResourceProperty`

In all the previous chapters, we used a special Globus-supplied class called `ReflectionResourceProperty` to implement our RPs. This class had a number of benefits, described in , the main one being that it greatly simplified our implementation since we could use the RPs as if they were normal Java variables.

In this chapter, however, we will use a different Globus-supplied class to represent our RPs: `SimpleResourceProperty`. Although we could also use `ReflectionResourceProperty`, this is a good chance to take a look at `SimpleResourceProperty`. Although this class makes the implementation a little bit more complicated, it shouldn't be too hard to understand now that we know what the *resource property document* is.

Note: Now is a good moment to review .

First of all, remember how in all the previous examples, our RPs were implemented simply as two attributes in the resource class:

```
private int value;
private String lastOp;
```

Now, this will be replaced by the following:

```
/* Resource properties */
private ResourceProperty valueRP;
private ResourceProperty lastOpRP;
```

> **Note:** `ResourceProperty` is a Globus-supplied interface that all resource properties must implement. Both `ReflectionResourceProperty` and `SimpleResourceProperty`, for example, implement it.

Furthermore, we are not *required* to implement get/set methods for the RPs, as we did when using `ReflectionResourceProperty`. However, as will be explained later on, it will nonetheless be convenient to do so, as it will make working with the RPs easier, specially if we split our implementation into a service, a resource, and a resource home.

Next, we need to initialize the resource properties. In our example, this is done in the constructor:

```
public MathService() throws RemoteException {
this.propSet = new SimpleResourcePropertySet(
MathQNames.RESOURCE_PROPERTIES); ❶

try {
valueRP = new SimpleResourceProperty(MathQNames.RP_VALUE); ❷
valueRP.add(new Integer(0)); ❸

❹
lastOpRP = new SimpleResourceProperty(MathQNames.RP_LASTOP);
lastOpRP.add("NONE");
} catch (Exception e) {
throw new RuntimeException(e.getMessage());
}

❺
this.propSet.add(valueRP);
this.propSet.add(lastOpRP);
}
```

❶ First, we create the RP set. This, in effect, creates an empty RP document. In our case, this would be something like this:

```
<MathResourceProperties xmlns:tns="http://www.globus.org/namespaces/examples/core/MathSe
</MathResourceProperties>
```

❷ Next, we create a `SimpleResourceProperty`. If you compare with the previous examples, you'll notice that the constructor for `SimpleResourceProperty` only requires the QName of the RP (whereas `ReflectionResourceProperty` required more parameters).

❸ Now, we set the initial value of the RP. When using `ReflectionResourceProperty`, we simply had to modify the `value` attribute. Now, however, we have to use `SimpleResourceProperty`'s `add` method to do this. By adding `new Integer(0)`, we are creating a new `Value` RP with value 0:

```
<tns:Value>0</tns:Value>
```

Note that, if the RP were unbounded, we could keep on invoking `valueRP.add` to create more `Value` RPs:

```
<tns:Value>0</tns:Value>
<tns:Value>0</tns:Value>
<tns:Value>0</tns:Value>
```

However, we cannot do this because `Value` is declared to occur once (and only once) in the RP document.

❹ We perform the previous two steps again to add a new `LastOp` RP.

❺ Finally, we add the RPs to the RP set. Now, our RP document will look something like this:

```
<MathResourceProperties xmlns:tns="http://www.globus.org/namespaces/examples/core/MathSe
<tns:Value>0</tns:Value>
<tns:LastOp>NONE</tns:LastOp>
</MathResourceProperties>
```

Publishing our RPs as topics with `ResourcePropertyTopic`

Now that our resource properties are implemented using the `SimpleResourceProperty` class, publishing those RPs as topics is very simple. We will use a Globus-supplied class called `ResourcePropertyTopic` which is both a resource property *and* a topic (more precisely, it implements both the `ResourceProperty` and *Topic* interfaces). As we will see, the only thing we need to to is create new `ResourcePropertyTopic` objects and "wrap them around" our `SimpleResourceProperty` objects. Then, the `ResourcePropertyTopic` objects are added both to the resource's list of RPs (the RP set) and the list of topics.

First of all, our resource class must implement the `TopicListAccessor` interface, which requires that we implement a `getTopicList` method returning a `TopicList`. A `TopicList` attribute must therefore be added to our resource class to keep track of all the topics published by our resource.

```
import org.globus.wsrf.TopicListAccessor;

public class MathService implements Resource, ResourceProperties,
TopicListAccessor {

private TopicList topicList;
```

```
// ...

/* Required by interface TopicListAccessor */
public TopicList getTopicList() {
return topicList;
}
}
```

Next, we initialize the topic list, create the `ResourcePropertyTopic` objects, and add the to the RP set and the topic list:

```
public MathService() throws RemoteException {
/* Create RP set */
this.propSet = new SimpleResourcePropertySet(
MathQNames.RESOURCE_PROPERTIES);

/* Initialize the RP's */
try {
valueRP = new SimpleResourceProperty(MathQNames.RP_VALUE);
valueRP.add(new Integer(0));

lastOpRP = new SimpleResourceProperty(MathQNames.RP_LASTOP);
lastOpRP.add("NONE");
} catch (Exception e) {
throw new RuntimeException(e.getMessage());
}

❶
this.topicList = new SimpleTopicList(this);

❷
valueRP = new ResourcePropertyTopic(valueRP);
((ResourcePropertyTopic) valueRP).setSendOldValue(true);

lastOpRP = new ResourcePropertyTopic(lastOpRP);
((ResourcePropertyTopic) lastOpRP).setSendOldValue(true);

❸
this.topicList.addTopic((Topic) valueRP);
this.topicList.addTopic((Topic) lastOpRP);

this.propSet.add(valueRP);
this.propSet.add(lastOpRP);
}
```

❶ We initialize the topic list using the Globus-supplied `SimpleTopicList` class.

❷ We take the previously created `SimpleResourceProperty` objects and put them "inside" `ResourcePropertyTopic` objects. Notice how the `valueRP` and `lastOpRP` attributes (of type `ResourceProperty`) are set to the `ResourcePropertyTopic` objects, *not* the original `SimpleResourceProperty` objects.

Next, we will activate a nice feature included in `ResourcePropertyTopic`s. We can ask that the notification include not only the new value (whenever an RP is modified), but also the old value.

❸ Finally, we add the `ResourcePropertyTopic` objects to the topic list.

> **Note:** The code shown above is part of `$EXAMPLES_DIR/org/globus/examples/services/core/notifications/impl/MathService.java`.

The service implementation

We need to modify the implementation of the `add` and `subtract` methods because we are now using `SimpleResourceProperty` objects to represent our RPs. However, take into account that none of the following changes are directly related to the fact that we're using notifications. When using `ResourcePropertyTopic`s, the notification is sent out *automatically* whenever we modify the value of an RP. We do not need to add any code to trigger the notification.

As mentioned earlier, using `SimpleResourceProperty` objects is going to make our interaction with the RP's a bit hairier. When using `ReflectionResourceProperty`, our `add` method was as simple as this:

```
public AddResponse add(int a) throws RemoteException {
value += a;
lastOp = "ADDITION";

return new AddResponse();
}
```

Now, however, the code will look something like this:

```
public AddResponse add(int a) throws RemoteException {
Integer value = (Integer) valueRP.get(0); ❶
value = new Integer(value.intValue()+a); ❷
valueRP.set(0, value); ❸
lastOpRP.set(0,"ADDITION"); ❹

return new AddResponse();
}
```

> **Note:** The code shown above is part of `$EXAMPLES_DIR/org/globus/examples/services/core/notifications/impl/MathService.java`.

❶ We retrieve the current value of the `Value` RP. Since this RP can have one (and only one) value, we have to access the value in position `0` using the `get` method. Since `get` returns an `Object`, we need to cast this into an `Integer` object.

❷ Next, we perform the actual addition.

❸ Next, we modify the value of the `Value` RP using the `set` method. Again, since this RP can hold a single value, the new value is placed in the first position of the RP (position 0).

❹ Finally, we modify the value of the `LastOp` RP using, once again, the `set` method.

Accessing the RPs if we split up the implementation: Remember that, in this example, we are using the `ServiceResourceHome` which allows us to implement the service and the resource in the same class. This means that our `add` and `subtract` methods have direct access to the `ResourceProperty` objects representing our RPs.

However, this will not be so if we split up the implementation as seen in and . In these cases, we will need to use our resource's `ResourceProperties` interface to access the RP set and then the RP's themselves. This means that our code could end up looking something like this:

```
public AddResponse add(int a) throws RemoteException {
MathResource mathResource = getResource();

ResourceProperty valueRP = mathResource.propSet.get(MathQNames.RP_VALUE);
Integer value = (Integer) valueRP.get(0);
value = new Integer(value.intValue()+a);
valueRP.set(0, value);

ResourceProperty lastOpRP = mathResource.propSet.get(MathQNames.RP_LASTOP);
lastOpRP.set(0,"ADDITION");

return new AddResponse();
}
```

Of course, this doesn't make things any nicer. This is why it is usually a good idea to include get/set methods for our RPs in the resource implementation, even if they are not required by `SimpleResourceProperty`. In other words, we could add the following to our resource:

```
public int getValue() {
Integer value_obj = (Integer) valueRP.get(0);
return value_obj.intValue();
}

public void setValue(int value) {
Integer value_obj = new Integer(value);
valueRP.set(0, value_obj);
}

public String getLastOp() {
String lastOp_obj = (String) lastOpRP.get(0);
return lastOp_obj;
}

public void setLastOp(String lastOp) {
lastOpRP.set(0, lastOp);
}
```

With these get/set methods, our `add` method could be implemented like this:

```
public AddResponse add(int a) throws RemoteException {
MathResource mathResource = getResource();

mathResource.setValue(mathResource.getValue() + a);
mathResource.setLastOp("ADDITION");

return new AddResponse();
}
```

This, of course, looks much nicer. In fact, it is very similar to the way we were able to implement `add` and `subtract` in and .

Deployment Descriptor

To be able to use the WS-Notifications portTypes we need to modify our WSDD file to make sure that our service relies on the Globus-supplied operation providers for those portTypes.

```
<?xml version="1.0" encoding="UTF-8"?>
<deployment name="defaultServerConfig"
    xmlns="http://xml.apache.org/axis/wsdd/"
    xmlns:java="http://xml.apache.org/axis/wsdd/providers/java"
    xmlns:xsd="http://www.w3.org/2001/XMLSchema">

    <service name="examples/core/notifications/MathService" provider="Handler" use="literal
        <parameter name="className" value="org.globus.examples.services.core.notifications.
        <wsdlFile>share/schema/examples/MathService_instance_notif/Math_service.wsdl</wsdlF
        <parameter name="allowedMethods" value="*"/>
        <parameter name="handlerClass" value="org.globus.axis.providers.RPCProvider"/>
        <parameter name="scope" value="Application"/>
        <parameter name="providers" value="GetRPProvider SubscribeProvider GetCurrentMessag
        <parameter name="loadOnStartup" value="true"/>
    </service>

</deployment>
```

Note: This file is `$EXAMPLES_DIR/org/globus/examples/services/core/notifications/deploy-server.wsdd`.

Compile and deploy

Let’s build the service:

```
./globus-build-service.sh notifications
```

And deploy it:

```
globus-deploy-gar $EXAMPLES_DIR/org_globus_examples_services_core_notifications.gar
```

Client code

To try out this service, we will need two clients. The first client will be in charge of *listening* for notifications, and includes a lot of new code. The second client is a very simple client that invokes the `add` operation. This will allow us to test if a change in the `Value` RP (triggered by the `add` operation) is indeed notified to the listener client.

Listener client

This client is composed of two important parts:

1. **Subscription**: This block of code will be in charge of setting up the subscription with the `Value` RP (which, remember, is also published as a topic). Once the subscription is set up, this block of code simply loops indefinitely.
2. **Delivery**: Once the subscription has been set up, and the main thread of the program is looping indefinitely, the delivery code gets invoked any time a notification arrives at the client. In fact, the listener must implement the `NotifyCallback` interface, which requires that we implement a `deliver` method that will be in charge of handling incoming notifications.

   ```
   public class ValueListener implements NotifyCallback {

   }
   ```

First off, let's take a look at the code that sets up the subscription. This code is inside a method called `run` that expects the notification producer's URI as its only parameter.

```
public void run(String serviceURI) {
try {
❶
NotificationConsumerManager consumer;

❷
consumer = NotificationConsumerManager.getInstance();
consumer.startListening();
EndpointReferenceType consumerEPR = consumer
.createNotificationConsumer(this);

❸
Subscribe request = new Subscribe();
request.setUseNotify(Boolean.TRUE);
request.setConsumerReference(consumerEPR);

❹
TopicExpressionType topicExpression = new TopicExpressionType();
topicExpression.setDialect(WSNConstants.SIMPLE_TOPIC_DIALECT);
```

```
topicExpression.setValue(MathQNames.RP_VALUE);
request.setTopicExpression(topicExpression);


❺
WSBaseNotificationServiceAddressingLocator notifLocator =
new WSBaseNotificationServiceAddressingLocator();
EndpointReferenceType endpoint = new EndpointReferenceType();
endpoint.setAddress(new Address(serviceURI));
NotificationProducer producerPort = notifLocator
.getNotificationProducerPort(endpoint);

❻
producerPort.subscribe(request);

❼
System.out.println("Waiting for notification. Ctrl-C to end.");
while (true) {
try {
Thread.sleep(30000);
} catch (Exception e) {
System.out.println("Interrupted while sleeping.");
}
}
} catch (Exception e) {
e.printStackTrace();
}
}
```

❶ Our client is going to act as a notification consumer. This means that our client will have to expose a `Notify` operation that will be invoked by the notification producer. For this to happen, our client has to act as both a client and a server. Fortunately, thanks to a Globus-supplied class called `NotificationConsumerManager`, we are shielded from all the potential nastiness involved in doing this is.

❷ Once we invoke the `startListening` method in the `NotificationConsumerManager`, our client becomes a server hosting a service that implements the standard `NotificationConsumer` portType. As such, this service will have an endpoint reference. We need to keep track of this EPR, since it will be used by the `NotificationProducer` to deliver the notifications.

❸ We create the request to the remote `Subscribe` call. There are two properties we must set: whether the producer must use the standard `Notify` operation to deliver notifications (in general, we will always want this to be true), and the consumer's EPR.

❹ Next, we create a `TopicExpressionType` object representing the topic we want to subscribe to. Notice how we're subscribing to the `Value` RP.

❺ A this point, the `Subscribe` request is ready to be sent to the notification producer. To do this, we need to obtain a reference to the standard `NotificationProducer` portType in the remote service.

❻ We are finally ready to send the subscription request.

❼ Finally, we let out program loop indefinitely. Notice that we instruct the client's thread to sleep during the loop. This doesn't affect the client's ability to receive notifications, as it will be woken up whenever a notification is delivered.

Now, let's take a look at the code that handles incoming notifications. Remember that the `delivery` is required by the `NotifyCallback` interface. If we do not implement it, our client will be unable to receive notifications.

```
public void deliver(List topicPath, EndpointReferenceType producer, ❶
Object message) {
ResourcePropertyValueChangeNotificationElementType notif_elem;
ResourcePropertyValueChangeNotificationType notif;

❷
notif_elem = (ResourcePropertyValueChangeNotificationElementType) message;
notif = notif_elem.getResourcePropertyValueChangeNotification();

if (notif != null) {
System.out.println("A notification has been delivered");

❸
System.out.print("Old value: ");
System.out.println(notif.getOldValue().get_any()[0].getValue());
System.out.print("New value: ");
System.out.println(notif.getNewValue().get_any()[0].getValue());
}
}
```

❶ The `deliver` method has three parameters:

1. `topicPath`: the topic that produced the notification.
2. `producer`: the EPR of the notification producer.
3. `message`: the actual notification. Notice how it is of type `Object`, so we will need to cast it to a more useful type.

❷ When using `ResourcePropertyTopic`s to notify changes in RPs, the notification message is of type `ResourcePropertyValueChangeNotificationElementType`. This type, in turn, contains an object of type `ResourcePropertyValueChangeNotificationType`. This object is the one that contains the new value of the RP. Remember that, in this example, we've also asked that the notification include the old value too.

❸ Finally, we print out the old and new values of the RP.

Note: The code shown above is part of
`$EXAMPLES_DIR/org/globus/examples/clients/MathService_instance_notif/ValueListener.java`

Adding client

The adding client requires no explanation, as it is identical to the ones seen in previous chapters.

> **Note:** The source code for the adding client is `$EXAMPLES_DIR/org/globus/examples/clients/MathService_instance_notif/ClientAdd.java`
>
> If you're not sure about how the client works, this might be a good time to review .

Compile and run

First of all, let's compile the listener client:

```
javac \
-classpath $CLASSPATH:build/stubs/classes/ \
org/globus/examples/clients/MathService_instance_notif/ValueListener.java
```

Since we are going to use two clients, you should run the listener in a separate console.

```
java \
-DGLOBUS_LOCATION=$GLOBUS_LOCATION \
-classpath $CLASSPATH:build/stubs/classes/ \
org/globus/examples/clients/MathService_instance_notif/ValueListener \
http://127.0.0.1:8080/wsrf/services/examples/core/notifications/MathService
```

> **Note:** Notice how we have to define a property called `GLOBUS_LOCATION`. This should be set to the directory where GT4 is installed. We need to define this property because, as mentioned earlier, our client is also going to act as a server. Therefore, it needs to know where all the Globus files are located (some of which are necessary for it to work as a server).

If all goes well, you should see the following:

```
Waiting for notification. Ctrl-C to end.
```

Now, let's compile the adder client:

```
javac \
-classpath $CLASSPATH:build/stubs/classes/ \
org/globus/examples/clients/MathService_instance_notif/ClientAdd.java
```

And run it:

```
java \
-classpath $CLASSPATH:build/stubs/classes/ \
org/globus/examples/clients/MathService_instance_notif/ClientAdd \
http://127.0.0.1:8080/wsrf/services/examples/core/notifications/MathService \
10
```

If all goes well, you should see the following:

```
Value RP: 10
LastOp RP: ADDITION
```

Now, if you check the console where the listener client is running, you should see the following:

```
A notification has been delivered
Old value:0
New value:10
```

You can try to run the adder client once more:

```
Value RP: 20
LastOp RP: ADDITION
```

And the following will be output by the listener.

```
A notification has been delivered
Old value:10
New value:20
```

III. GT4 Security

Before reading this part of the tutorial...

First of all, if you've been reading the tutorial from the beginning, and have successfully tried out all the examples, then it's time to sit back for a second and give yourself a pat on the back!

Ready to continue? We are now entering the next major part of this tutorial: **GT4 Security**. This part of the tutorial assumes that the reader knows his way around GT4 Java WS Core and all the fundamental concepts (how to compile a service, how to deploy it, etc.). This means some explanations won't be as detailed (to avoid being repetitious). One of the first things you'll notice is that, since the examples are starting to be quite long, complete code listings will be less frequent. Instead, relevant code sections will be described. Therefore, you'll need to download the complete code of the examples from the tutorial website (http://gdp.globus.org/gt4-tutorial/) to try out the services by yourself.

Chapter 9. Fundamental Security Concepts

Working with the security components of GT4 requires, of course, a basic knowledge of certain fundamental computer security concepts. If you are already familiar with concepts such as authentication, authorization, public key cryptography, and certificate authorities, then you can safely skip this chapter. If you've never dealt with secure communications, or feel your knowledge of these concepts might be a bit rusty, then you should definitely read this chapter. However, take into account that this chapter is meant as an *overview* of these concepts. Some readers, specially complete newcomers, should consider reading some material that deals specifically with computer security:

- *Practical Cryptography*. Bruce Schneier. John Wiley & Sons, 2003. http://www.schneier.com/book-practical.html.
- *Applied Cryptography*. Bruce Schneier. John Wiley & Sons, 1996. http://www.schneier.com/book-applied.html.

What is a secure communication?

The first thing we have to ask ourselves is: Well, just what *is* a secure communication? Newcomers to the field of computer security tend to think that a 'secure communication' is simply any communication where data is encrypted. However, security encompasses much more than simply encrypting and decrypting data.

The Three Pillars of a Secure Communication

Most authors consider the three pillars of a secure communication (or 'secure conversation') to be *privacy*, *integrity*, and *authentication*. Ideally, a secure conversation should feature all three pillars, but this is not always so (sometimes it might not even be desirable). Different security scenarios might require different combination of features (e.g. "only privacy", "privacy and integrity, but no authentication", "only integrity", etc.).

> **Note:** You might stumble upon books and URLs which also talk about 'non-repudiation', a feature which some authors consider the 'fourth pillar' of secure conversations. Since non-repudiation never comes up in Globus literature, and because most authors tend to simply consider it a part of 'authentication', we've chosen not to include it in this chapter.

Privacy

A secure conversation should be *private*. In other words, only the sender and the receiver should be able to understand the conversation. If someone eavesdrops on the communication, the eavesdropper should be unable to make any sense out of it. This is generally achieved by encryption/decryption algorithms.

For example, imagine we want to transmit the message "INVOKE METHOD ADD", and we want to make sure that, if a third party intercepts that message (e.g. using a network sniffer), they won't be able to understand that message. We could use a trivial encryption algorithm which simply changes each

letter for the next one in the alphabet. The encrypted message would be "JOWPLFANFUIPEABEE" (let's suppose 'A' comes after the whitespace character). Unless the third party knew the encryption algorithm we're using, the message would sound like complete gibberish. On the other hand, the receiving end would know the decryption algorithm beforehand (change each letter for the *previous* one in the alphabet) and would therefore be able to understand the message. Of course, this method is trivial, and encryption algorithms nowadays are much more sophisticated. We'll look at some of those algorithms in the next section.

Integrity

A secure communication should ensure the *integrity* of the transmitted message. This means that the receiving end must be able to know *for sure* that the message he is receiving is exactly the one that the transmitting end sent him. Take into account that a malicious user could intercept a communication with the intent of modifying its contents, not with the intent of eavesdropping.

'Traditional' encryption algorithms don't protect against these kind of attacks. For example, consider the simple algorithm we've just seen. If a third party used a network sniffer to change the encrypted message to "JAMJAMJAMJAMJAMJA", the receiving end would apply the decryption algorithm and think the message is "I LI LI LI LI LI ". Although the malicious third party might have no idea what the message contains, he is nonetheless able to modify it (this is relatively easy to do with certain network sniffing tools). This confuses the receiving end, which would think there has been an error in the communication. Public-key encryption algorithms (which we'll see shortly) *do* protect against this kind of attacks (the receiving end has a way of knowing if the message it received is, in fact, the one the transmitting end sent and, therefore, not modified).

Authentication

A secure communication should ensure that the parties involved in the communication are who they claim to be. In other words, we should be protected from malicious users who try to *impersonate* one of the parties in the secure conversation. Again, this is relatively easy to do with some network sniffing tools. However, modern encryption algorithms also protect against this kind of attacks.

Authorization

Another important concept in computer security, although not generally considered a 'pillar' of secure communications, is the concept of *authorization*. Simply put, authorization refers to mechanisms that decide when a user is *authorized* to perform a certain task. Authorization is related to authentication because we generally need to make sure that a user is who he claims to be (authentication) before we can make a decision on whether he can (or cannot) perform a certain task (authorization).

For example, once we've ascertained that a user is a member of the Mathematics Department, we would then allow him to access all the MathServices. However, we might deny him access to other services that are not related to his department (BiologyService, ChemistryService, etc.)

> **Authorization vs. Authentication:** It is very easy to confuse *authentication* and *authorization*, not so much because they are related (you generally need to perform authentication on a user to make

authorization decisions on that user), but because they sound alike! ("auth...ation") This is somewhat aggravated by the fact that many people tend to shorten both words as "auth" (especially in programming code). At this point, you might be saying to yourself: "That's pretty silly, they're different concepts... I'm not going to confuse them just because they sound alike!" Well, believe me, it happens, and quite a lot :-) When in doubt, remember that *authentication* refers to finding out if someone's identity is *authentic* (if they really are who they claim to be) and that *authorization* refers to finding out is someone is *authorized* to perform a certain task.

Introduction to cryptography

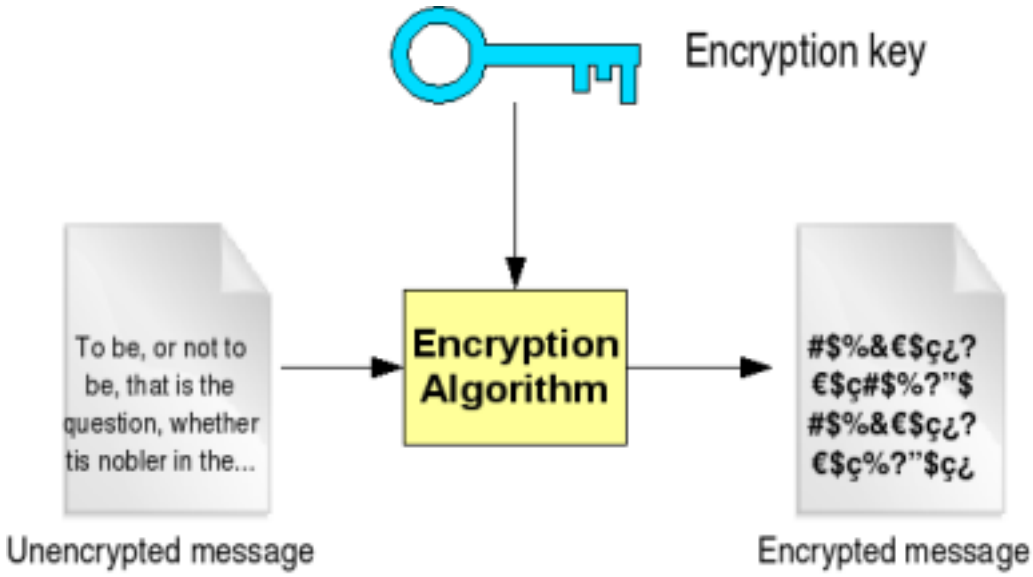
Cryptography is "the art of writing in secret characters". *Encrypting* is the act of translating a 'normal message' to a message written with 'secret characters' (also known as the *encrypted message*). Decrypting is the act of translating a message written with 'secret characters' into a readable message (the *unencrypted message*). It is, by far, one of the most important areas in computer security, since modern encryption algorithms can ensure all three pillars of a secure conversation: privacy, integrity, and authentication.

Key-based algorithms

In the previous page we saw a rather simple encryption algorithm which simply substituted each letter in a message by the next one in the alphabet. The decryption algorithm was, of course, substituting each letter in the encrypted message with the *previous* letter in the alphabet. These kind of algorithms, based on the *substitution* of letters, are easily broken. Most modern algorithms, however, are *key-based.*

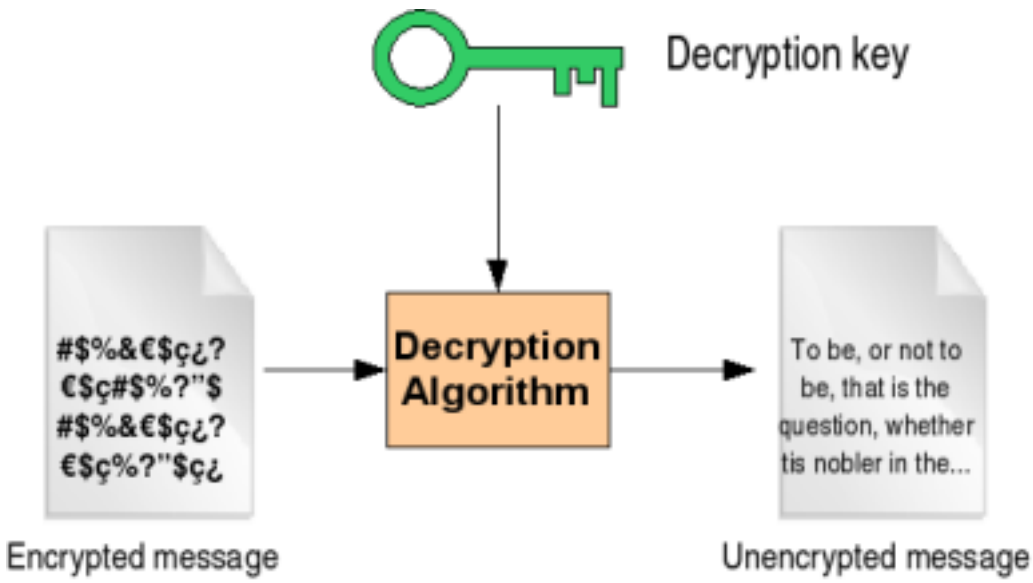
A *key-based algorithm* uses an *encryption key* to encrypt the message. This means that the encrypted message is generated using not only the message, but also using a 'key':

Figure 9-1. Key-based encryption



The receiver can then use a *decryption key* to decrypt the message. Again, this means that the decryption algorithm doesn't rely only on the encrypted message. It also needs a 'key':

Figure 9-2. Key-based decryption



Some algorithms use the same key to encrypt and decrypt, and some do not. However, we'll look into this in more detail in the next page.

Let's take a look at a simple example. To make things simpler, let's suppose we're not transmitting alphanumerical characters, only numerical characters. For example, we might be interested in transmitting the following message:

```
1 2 3 4 5 6 5 4 3 2 1
```

We will now choose a key which will be used to encrypt the message. Let's suppose the key is "4232". To encrypt the message, we'll repeat the key as many times as necessary to 'cover' the whole message:

```
1 2 3 4 5 6 5 4 3 2 1

4 2 3 2 4 2 3 2 4 2 3
```

Now, we arrive at the encrypted message by adding both numbers:

```
  1 2 3 4 5 6 5 4 3 2 1

+ 4 2 3 2 4 2 3 2 4 2 3
  ---------------------
  5 4 6 6 9 8 8 6 7 4 4
```

The resulting message (54669886744) is the encrypted message. We can decrypt following the inverse process: Repeating the key as many time as necessary to cover the message, and then *subtract* the key character by character:

```
  5 4 6 6 9 8 8 6 7 4 4

- 4 2 3 2 4 2 3 2 4 2 3
  ---------------------
  1 2 3 4 5 6 5 4 3 2 1
```

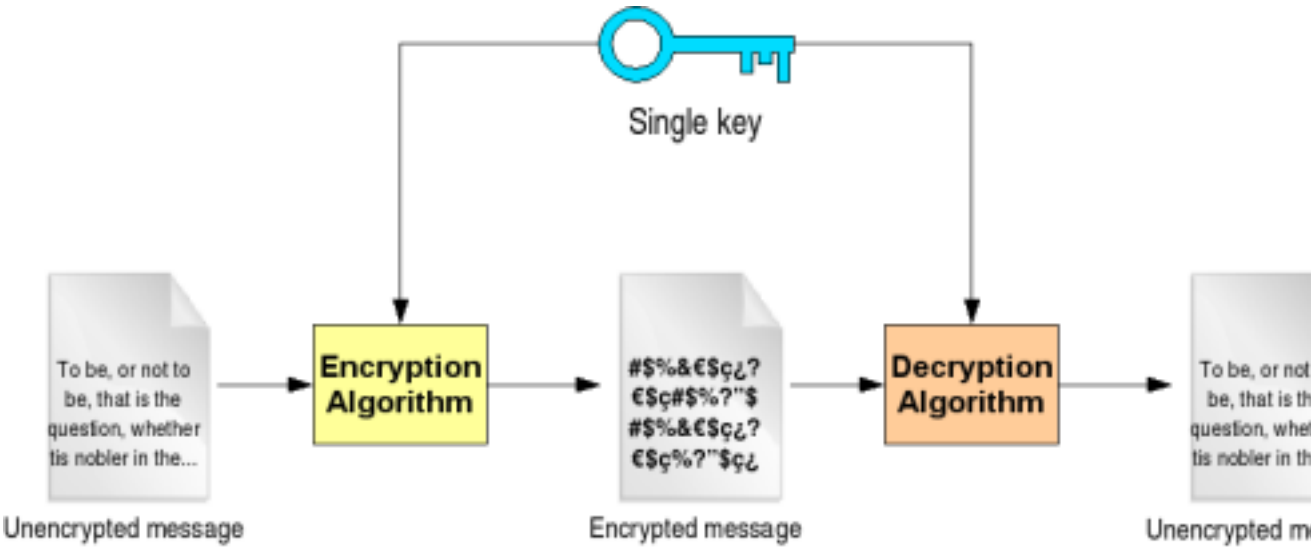
Voilà! We're back at the unencrypted message! Notice how it is absolutely necessary to have the decryption key (in this case, the same as the encryption key) to be able to decrypt the message. This means that a malicious user would need both the message *and* the key to eavesdrop on our conversation.

Please note that this is a very trivial example. Current key-based algorithms are *much more* sophisticated (for starters, keys are much longer, and the encryption process is not as simple as 'adding the message and the key'). However, these complex algorithms *are based* on the same basic principle shown in our example: a key is needed to encrypt/decrypt message.

Symmetric and asymmetric key-based algorithms

The example algorithm we've just seen falls into the category of *symmetric algorithms*. These type of algorithm uses *the same key* for encryption and decryption:

Figure 9-3. Key-based symmetric algorithm



Although this type of algorithms are generally very fast and simple to implement, they also have several drawbacks. The main drawback is that they only guarantee privacy (integrity and authentication would have the be done some other way). Another drawback is that both the sender and the receiver need to agree on the key they will use throughout the secure conversation (this is not a trivial problem).

Secure systems nowadays tend to use *asymmetric algorithms*, where a different key is used to encrypt and decrypt the message. *Public-key algorithms*, which are introduced in the next section, are the most commonly used type of asymmetric algorithms.

Public key cryptography

Public-key algorithms are *asymmetric* algorithms and, therefore, are based on the use of two different keys, instead of just one. In public-key cryptography, the two keys are called the *private key* and the *public key*

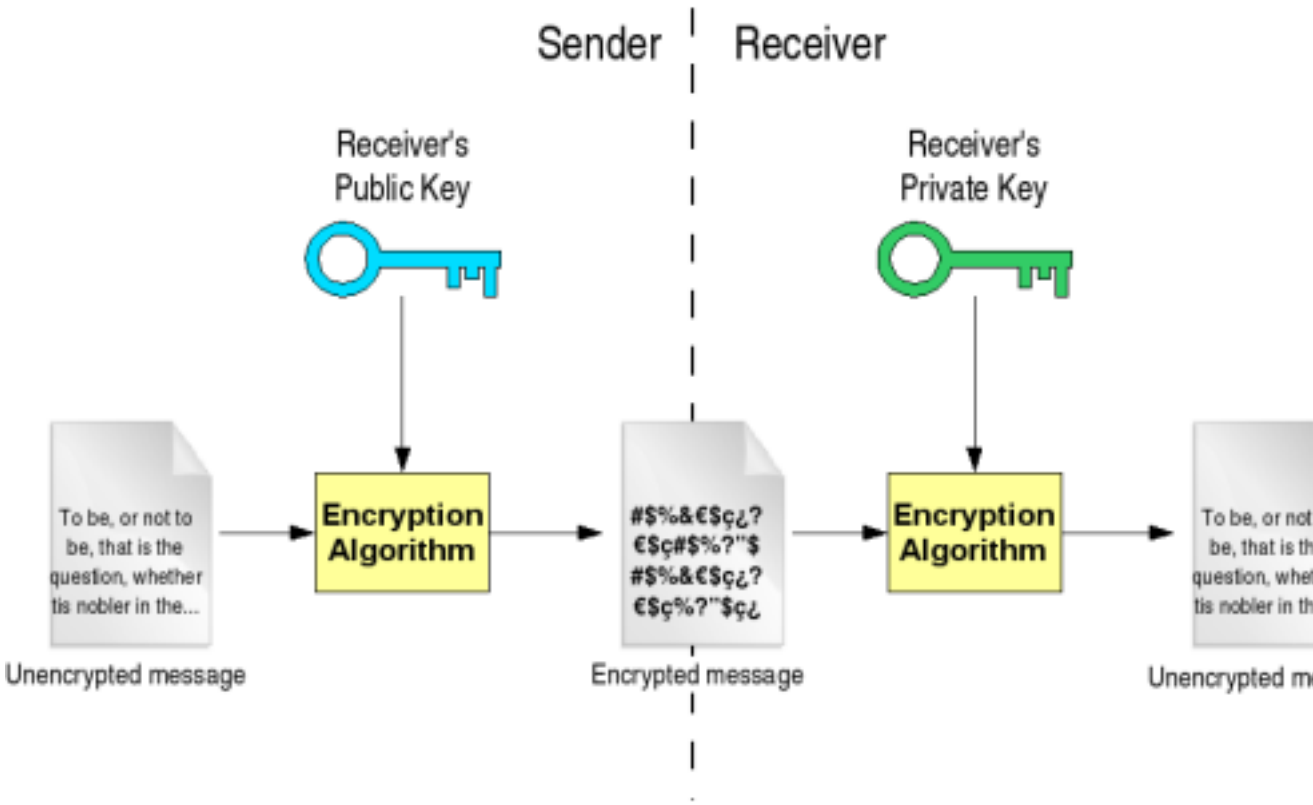
- **Private key**: This key must be know *only* by its owner.
- **Public key**: This key is known to everyone (it is *public*)
- **Relation between both keys**: What one key encrypts, the other one decrypts, and vice versa. That means that if you encrypt something with my public key (which you would know, because it's public :-), I would need my private key to decrypt the message.

A secure conversation using public-key cryptography

In a basic secure conversation using public-key cryptography, the sender encrypts the message using the

receiver's *public* key. Remember that this key is known to everyone. The encrypted message is sent to the receiving end, who will decrypt the message with his *private* key. Only the receiver can decrypt the message because no one else has the private key. Also, notice how the encryption algorithm is the same at both ends: what is encrypted with one key is decrypted with the other key using the same algorithm.

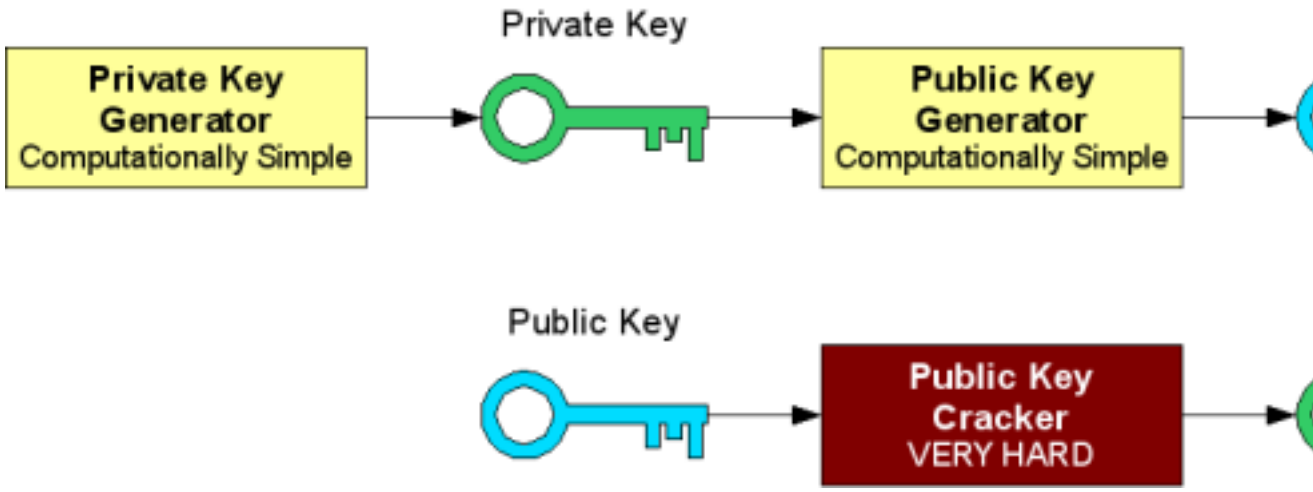
Figure 9-4. Key-based asymmetric algorithm



Pros and cons of public-key systems

Public-key systems have a clear advantage over symmetric algorithms: there is no need to agree on a common key for both the sender and the receiver. As seen in the previous example, if someone wants to receive an encrypted message, the sender only needs to know the receiver's public key (which the receiver will provide; publishing the *public* key in no way compromises the secure transmission). As long as the receiver keeps the private key secret, no one but the receiver will be able to decrypt the messages encrypted with the corresponding public key. This is due to the fact that, in public-key systems, it is relatively easy to compute the public key from the private key, but *very hard* to compute the private key from the public key (which is the one everyone knows). In fact, some algorithms need several *months* (and even years) of constant computation to obtain the private key from the public key.

Figure 9-5. Public key generation



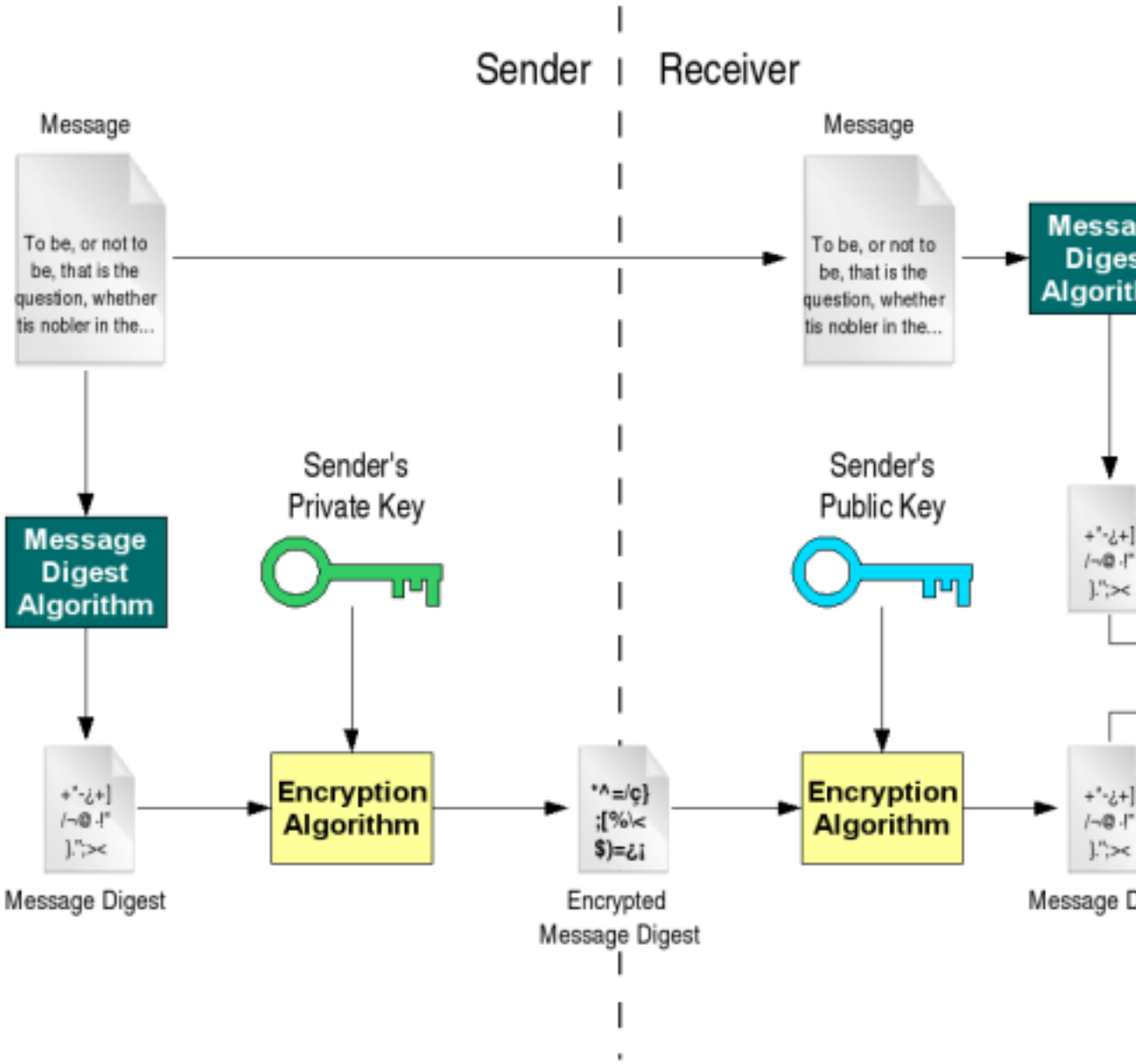
Another important advantage is that, unlike symmetric algorithms, public-key systems can guarantee integrity and authentication, not only privacy. The basic communication seen above only guarantees privacy. We will shortly see how integrity and authentication fit into public-key systems.

The main disadvantage of using public-key systems is that they are not as fast as symmetric algorithms.

Digital signatures: Integrity in public-key systems

Integrity is guaranteed in public-key systems by using *digital signatures*. A digital signature is a piece of data which is attached to a message and which can be used to find out if the message was tampered with during the conversation (e.g. through the intervention of a malicious user)

Figure 9-6. Digital signatures



The digital signature for a message is generated in two steps:

1. A *message digest* is generated. A message digest is a 'summary' of the message we are going to transmit, and has two important properties: (1) It is always smaller than the message itself and (2) Even the slightest change in the message produces a different digest. The message digest is generated using a set of hashing algorithms.
2. The message digest is encrypted using the sender's *private* key. The resulting encrypted message digest is the *digital signature*.

The digital signature is attached to the message, and sent to the receiver. The receiver then does the following:

1. Using the sender's public key, decrypts the digital signature to obtain the message digest generated by the sender.
2. Uses the same message digest algorithm used by the sender to generate a message digest of the received message.
3. Compares both message digests (the one sent by the sender as a digital signature, and the one generated by the receiver). If they are not *exactly the same*, the message has been tampered with by a third party. We can be sure that the digital signature was sent by the sender (and not by a malicious user) because *only* the sender's public key can decrypt the digital signature (which was encrypted by the sender's private key; remember that what one key encrypts, the other one decrypts, and vice versa). If decrypting using the public key renders a faulty message digest, this means that either the message or the message digest are not exactly what the sender sent.

Using public-key cryptography in this manner ensures integrity, because we have a way of knowing if the message we received is exactly what was sent by the sender. However, notice how the above example guarantees *only* integrity. The message itself is sent unencrypted. This is not necessarily a bad thing: in some cases we might not be interested in keeping the data private, we simply want to make sure it isn't tampered with. To add privacy to this conversation, we would simply need to encrypt the message as explained in the first diagram.

Authentication in public-key systems

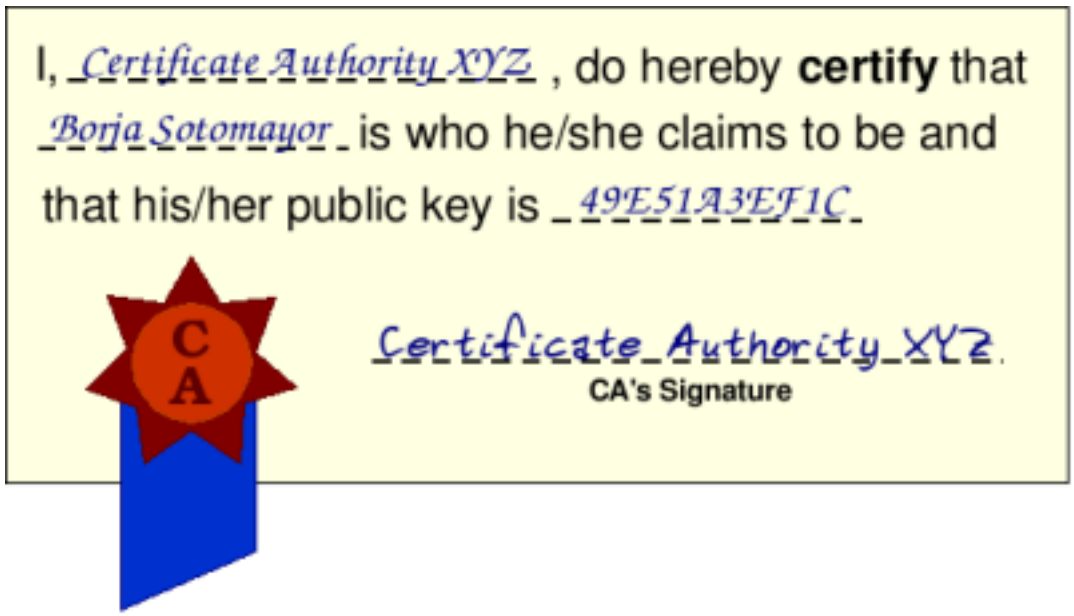
The above example does guarantee, to a certain extent, the authenticity of the sender. Since *only* the sender's public key can decrypt the digital signature (encrypted with the sender's *private* key). However, the only thing this guarantees is that whoever sent the message has the private key corresponding to the public key we used to decrypt the digital signature. Although this public key might have been advertised as belonging to the sender, how can we be absolutely certain? Maybe the sender isn't really who he claims to be, but just someone impersonating the sender.

Some security scenarios might consider that the 'weak authentication' shown in the previous example is sufficient. However, other scenarios might require that there is absolutely no doubt about a user's identity. This is achieved with *digital certificates*, which are explained in the next page.

Certificates and certificate authorities

A *digital certificate* is a digital document that *certifies* that a certain public key is owned by a particular user. This document is signed by a third party called the *certificate authority* (or CA). Figure 9-7 might help you get an idea of what a digital certificate is.

Figure 9-7. A digital certificate



Of course, the certificate is encoded in a digital format (no, you don't get a paper diploma so you can brag to your pals that "you really are who you claim to be" :-) The important thing to remember is that the certificate is *signed* by a third party (the certificate authority) which does not itself take place in the secure conversation. The signature is actually a digital signature generated with the CA's private key. Therefore, we can verify the integrity of the certificate using the CA's public key.

It's all about trust

Having a certificate to prove to everyone else that your public key is really, truly, honestly yours allows us to conquer the third pillar of a secure conversation: authentication. If you digitally sign your message with your private key, and send the receiver a copy of your certificate, he can know for sure that the message was sent by *you* (because only your public key can decrypt the digital signature... and the certificate assures that the public key the receiver uses is yours and no one else's)

However, all this is true supposing you *trust* the certificate. To be more exact, you have to *trust the CA that signs the certificate*. Believe it or not, there are no fancy algorithms to decide when a CA is trustworthy... you must decide by yourself whether you trust or don't trust a CA. This means that the public-key system you use will generally have a list of 'trusted CAs', which includes the digital certificates of those CAs you will trust (each of these certificates, in turn, include the CA's public key, so you can verify digital signatures).

You have to decide which CAs make it into the list. Some CAs are so well known that they are included by default in many public-key systems (for example, web browsers usually include VeriSign (http://www.verisign.com) and GlobalSign (http://www.globalsign.com) certificates, because many websites use certificates issued by those companies to authenticate themselves to web browsers). Of course, you can add other CAs to the 'trusted list'. For example, if your department sets up a CA, and you *trust* that the department's CA will only issue certificates to trustworthy people, then you could add it to the list.

X.509 certificate format

Now that we've gone through the basics, let's take a look at the format in which digital certificates are encoded: the X.509 certificate format. An X.509 certificate is a plain text file which includes a lot of information in a very specific syntax. That syntax is beyond the scope of this document, and we'll simply mention the four most important things we can find in an X.509 certificate:

- **Subject**: This is the 'name' of the user. It is encoded as a *distinguished name* (the format for distinguished names will be explained next)
- **Subject's public key**: This includes not only the key itself, but information such as the algorithm used to generate the public key.
- **Issuer's Subject**: CA's distinguished name.
- **Digital signature**: The certificate includes a digital signature of all the information in the certificate. This digital signature is generated using the CA's private key. To verify the digital signature, we need the CA's public key (which can be found in the CA's certificate).

As you can see, the information we can find in an X.509 certificate is the same which was shown in the illustration at the beginning of this page (name, CA's name, public key, CA's signature). Notice how the certificate, however, does *not* include the private key, which must be kept separate from the public key. Remember that the certificate is a public document we want to be able to distribute to other users so they can verify our identity, so we don't want to include the private key (which must be known only by the owner of the certificate). When we are in possesion of both a certificate and its associated private key, these two items are generally refered to as the user's `credentials`.

Distinguished names

Names in X.509 certificates are not encoded simply as 'common names', such as "Borja Sotomayor", "Lisa Childers", "Certificate Authority XYZ", or "Systems Administrator". They are encoded as *distinguished names*, which are a comma-separated list of name-value pairs. For example, the following could be our distinguished names:

```
O=University of Chicago, OU=Department of Computer Science, CN=Borja Sotomayor

O=Argonne National Laboratory, OU=Mathematics and Computer Science Division, CN=Lisa Childe
```

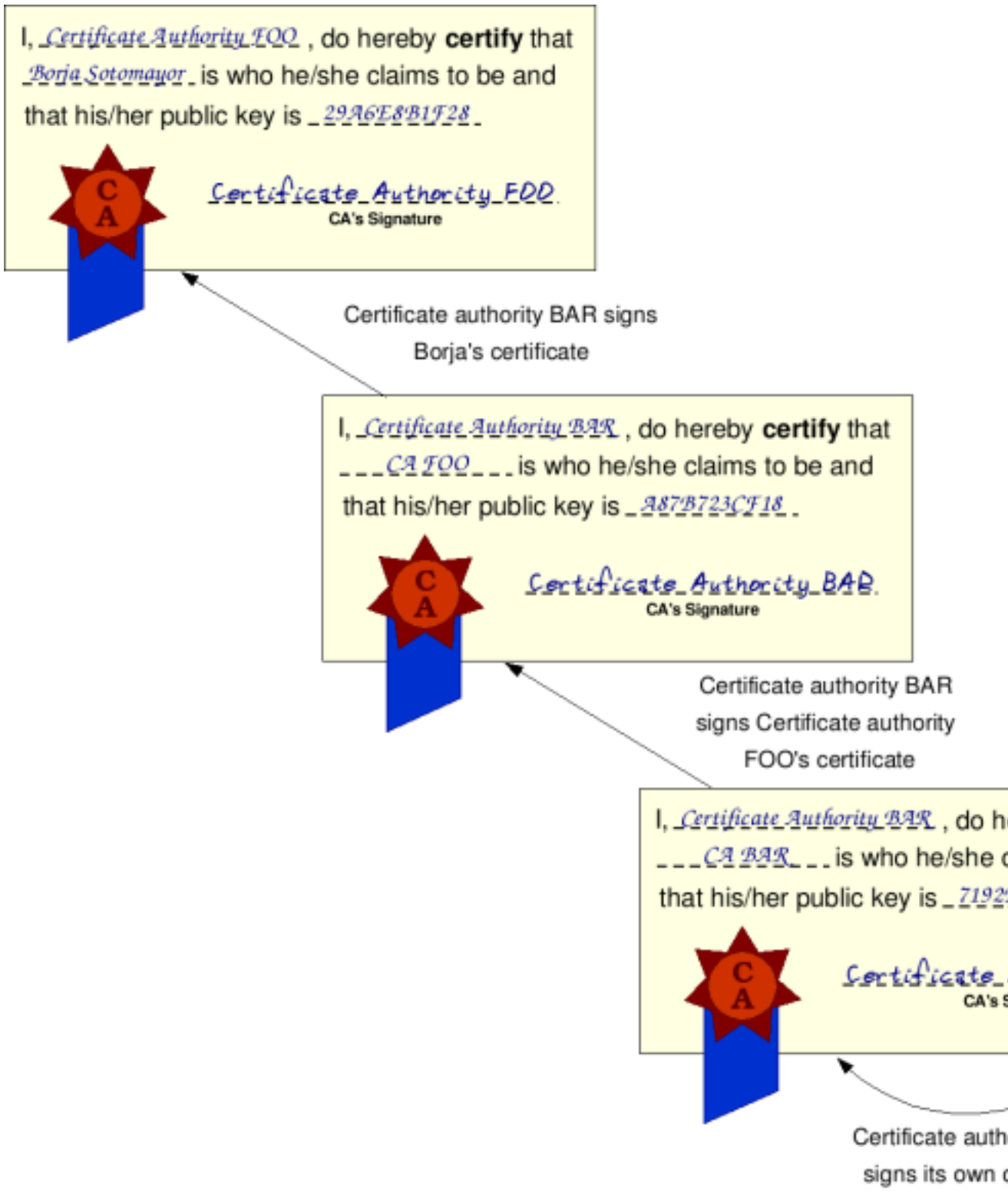
So what do "O", "OU", and "CN" mean? A distinguished name can have several different attributes, and the most common are the following:

- **O** : Organization
- **OU** : Organizational Unit
- **CN** : Common Name (generally, the user's name)
- **C** : Country

CA hierarchies

We mentioned earlier that your ’trusted CA list’ includes the certificates of all the CAs you decided to trust. At that point, you might have asked yourself: And who signs the CA’s certificate? The answer is very simple: Another CA! This allows for hierarchies of CAs to be created, in such a way that although you might not explicitly trust a CA (because it’s not in your list), you might trust the higher-level CA that signed its certificate (which makes the lower-level CA trustworthy). Figure 9-8 might make things a bit clearer.

Figure 9-8. Digital certificate chain of verification



In the illustration, Borja's certificate is signed by Certificate Authority FOO. Certificate Authority FOO's certificate is, in turn, signed by Certificate Authority BAR. Finally, BAR's certificate is signed by itself (we'll get to this in a second).

If you receive Borja's certificate, and don't explicitly trust CA FOO (the issuer of my certificate), this doesn't automatically mean the certificate isn't trustworthy. You might check to see if CA FOO's certificate was issued by a CA you *do* trust. If it turns out that CA BAR is in your 'trusted list', then that means that Borja's certificate is trustworthy.

However, notice that the higher-level CA (BAR) has signed its own certificate. This is not uncommon, and is called a *self-signed certificate*. A CA with a self-signed certificate is called a *root CA*, because there's 'no one above it'. To trust a certificate signed by this CA, it must necessarily be in your 'trusted CA list'.

Chapter 10. GSI: Grid Security Infrastructure

This chapter introduces the Grid Security Infrastructure, the basis for GT4's Security layer. A working knowledge of fundamental security concepts is assumed in this chapter. If you've read the previous chapter, you should be fine. If you haven't, but you know how public-key cryptography, certificates, and certificate authorities work, then you should also be fine.

Introduction to GSI

If you're familiar with Grid Computing, you probably know that security is one of the most important parts of a Grid application. Since a grid implies crossing organizational boundaries, resources are going to be accessed by a lot of different organizations. This poses a lot of challenges:

- We have to make sure that only certain organizations can access our resources, and that we're 100% sure that those organizations are really who they claim to be. In other words, we have to make sure that everyone in our grid application is properly *authenticated*.
- We're going to bump into some pretty interesting scenarios. For example, suppose organization AliceOrg asks BobOrg to perform a certain task. BobOrg, on the other hand, realizes that the task should be *delegated* to organization CharlieOrg. However, let's suppose CharlieOrg only trusts AliceOrg (and not BobOrg). Should CharlieOrg turn down the request because it comes from BobOrg, or accept it since the 'original' requestor is AliceOrg?
- Depending on our application, we may also be interested in assuring data *integrity* and *privacy*, although in a grid application this is generally not as important as authentication.

The Globus Toolkit 4 allows us to overcome the security challenges posed by grid applications through the *Grid Security Infrastructure* (or GSI). GSI is composed of a set of command-line tools to manage certificates, and a set of Java classes to easily integrate security into our web services. GSI offers programmers the following features, which we will discuss in the next sections:

- Transport-level and message-level security
- Authentication through X.509 digital certificates
- Several authorization schemes
- Credential delegation and single sign-on
- Different levels of security: container, service, and resource

Transport-level and message-level security

GSI allows us to enable security at two levels: the *transport* level or the *message* level. To explain the difference between these two levels, let's suppose we want our communication to be private. If we use transport-level security, as shown in Figure 10-1, then the complete communication (all the information exchanged between the client and the server) would be encrypted. If we use message-level security, as

shown in Figure 10-2, then only the *content* of the SOAP message is encrypted, while the rest of the SOAP message is left unencrypted.

Figure 10-1. Transport-level security

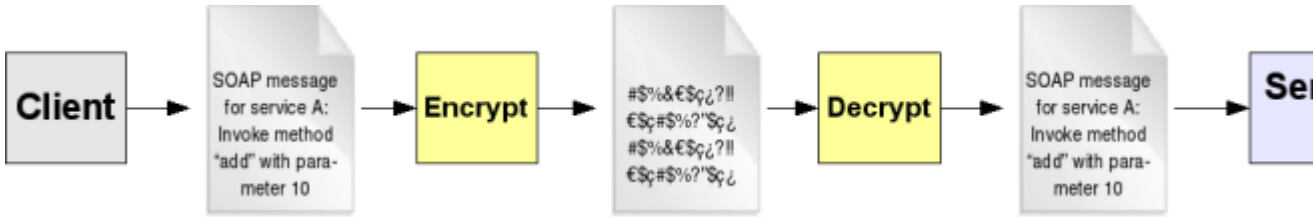


Figure 10-2. Message-level security

Both transport-level and message-level security in GSI are based on public-key cryptography and, therefore, can guarantee privacy, integrity, and authentication. However, not all communications need to have those three features all at once. In general, a GSI secure conversation must *at least* be authenticated. Integrity is usually desirable, but can be disabled. Encryption can also be activated to ensure privacy. As soon as we start programming secure services, we'll see how using these features is as easy as adding a few lines in the client indicating that (for example) we want to use integrity, but not encryption during the communication.

> **Message-level vs. Transport-level performance:** Transport-level security has been around for a long time and, in fact, chances are that you've already used it when browsing the Web, since secure websites rely on transport-level security. Message-level security in Web Services is relatively new and, although it offers more features than transport-level security (e.g. a better integration with Web Services standards), its performance still leaves a bit to be desired. So, even though we would ideally like to use message-level security for everything (because of its feature-rich goodness), we will sometimes have to consider using transport-level security if performance is an issue. In fact, transport-level security is used by default in the Globus Toolkit.

GSI offers two message-level protection schemes, and one transport-level scheme. The differences between these three schemes are highlighted in Table 10-1.

- **GSI Secure Message**: Provides message-level security and is based on the proposed WS-Security standard.
- **GSI Secure Conversation**: Provides message-level security and is based on the WS-SecureConversation specification. When this method is chosen, a *secure context* is first

established between the client and the server. After an initial exchange of messages to establish the context, all the following messages can reuse that context, resulting in a better performance than GSI Secure Message (if the overhead of setting up the context is acceptable). Furthermore, GSI Secure Conversation is the only scheme that supports credential delegation (explained further on).

- **GSI Transport**: Provides transport-level security by using TLS (formerly known as SSL). It provides the best performance and is used by default in GT4.

These schemes are *not* mutually exclusive. For example, we might choose to use GSI Secure Conversation because our application requires delegation, and then add GSI Transport on top of that because we want to encrypt the complete communication (not just a part of the SOAP message). Note that this doesn't result in any redundancy, since we could configure GSI Transport to use encryption and GSI Secure Conversation to *not* use encryption.

Table 10-1. Comparison of transport-level and message-level security

	GSI Secure Conversation	GSI Secure M
Technology	WS-SecureConversation	WS-Secu
Privacy (Encrypted)	YES	YES
Integrity (Signed)	YES	YES
Anonymous authentication	YES	NO
Delegation	YES	NO
Performance	Good if sending many messages	Good if sending f

Authentication

GSI supports three authentication methods:

- **X.509 certificates**: All three protection schemes seen above can be used along with X.509 certificated to provide strong authentication (as seen in).
- **Username and password**: A more rudimentary form of authentication, using usernames and passwords, can also be used. However, when using usernames and password, we will not be able to use features like privacy, integrity, and delegation. This form of authentication is not covered in the tutorial (you can refer to the official Globus documentation for more details on how to use it).
- **Anonymous authentication**: We can request that a communication be anonymous, or *unauthenticated*. Anonymous generally makes sense when we are using more than one security scheme. For example, we can use GSI Secure Conversation (authenticated with X.509 certificates) and anonymous GSI Transport, so that we don't perform an additional (redundant) authentication.

> **Note:** Since unauthenticated communications are not commonly used, the Globus literature generally uses the term *authentication methods* to refer directly to GSI Secure Conversation, GSI Secure Message, and GSI Transport. We will follow this same convention throughout the rest of the tutorial.

Authorization

Although authorization is not one of the ’fundamental pillars’ of a secure conversation, it is nonetheless an important part of GSI. Authorization refers to who is *authorized* to perform a certain task. In a Web services context, we will generally need to know who is authorized to use a certain web service.

GSI supports authorization in both the server-side and the client-side. Several authorization mechanisms are already included with the toolkit, but we will also be able to implement our own authorization mechanisms.

Server-side authorization

The server has six possible authorization modes. Depending on the authorization mode we choose, the server will decide if it accepts or declines an incoming request.

- **None**: This is the simplest type of authorization. No authorization will be performed.
- **Self**: A client will be allowed to use a service if the client’s identity is the same as the service’s identity.
- **Gridmap**: A gridmap is a list of ’authorized users’ akin to an ACL (Access Control List). . When this type of authorization is used, only the users that are listed in the service’s gridmap may invoke it.
- **Identity authorization**: A client will be allowed to access a service if the client’s identity matches a specified identity. In a sense, this is like having a one-user gridmap (except that identity authorization is configured programmatically, whereas the gridmap is represented as a file in our system).
- **Host authorization**: A client will be allowed to access a service if it presents a host credential that matches a specified hostname. In other words, we will only allow requests coming from one particular host.
- **SAML Callout authorization**: We can delegate the authorization decision to an OGSA Authorization-compliant authorization service. OGSA-Authz (http://forge.gridforum.org/projects/ogsa-authz) is a GGF working group whose goal is to specify standard authorization components. One of the main technologies used in these components is SAML (Security Assertion Markup Language).

Client-side authorization

This allows the client to figure out when it will allow a service to be invoked. This might seem like an odd type of authorization, since authorization is generally seen from the server’s perspective ("Do I allow client FOO to connect to grid service BAR?"). However, in GSI, clients have every right to be picky about the services they can access.

- **None**: No authorization will be performed.
- **Self**: The client will authorize an invocation if the service’s identity is the same as the client.
- **Identity authorization**: As described above, the client will only allow requests to be sent to services with a specified identity.
- **Host**: The client will authorize an invocation if the service has a host credential. Furthermore, the client must be able to resolve the address of the host to the hostname specified in the host credential.

Note that this is different from server-side host authorization, where we check if the hostname in the credential is equal to a host specified by us.

Custom authorization

GSI provides an infrastructure to easily plug in our own authorization mechanisms. For example, our organization might be using a legacy authorization service that can't work out-of-the-box with the authorization methods provided by the toolkit. In this case, we can create a new authorization method that will allow GSI to make authorization decisions based on our organization's legacy service.

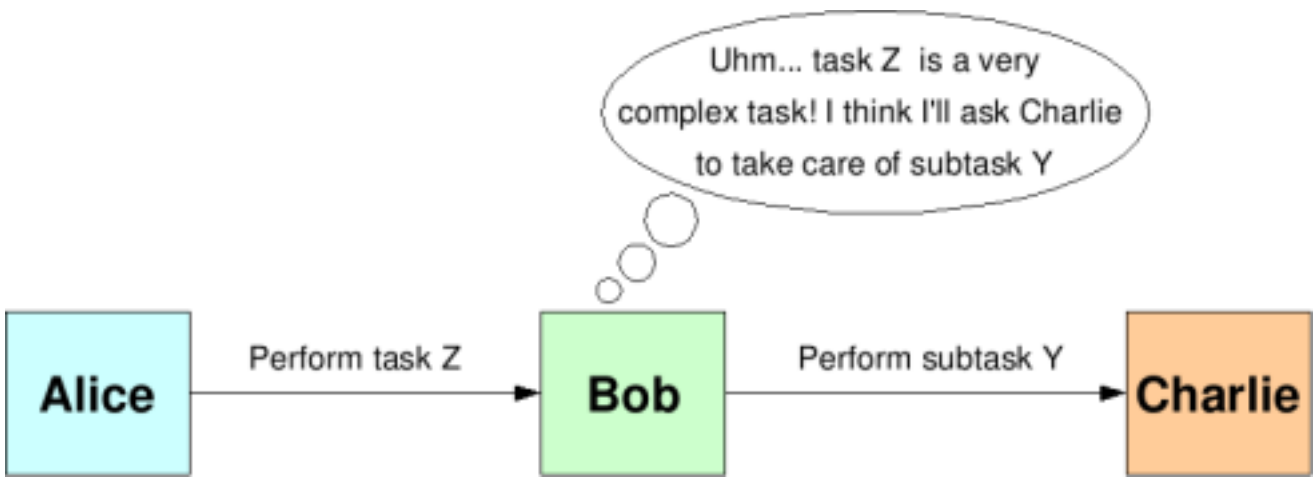
Delegation and single sign-on (proxy certificates)

Credential delegation and single sign-on are one of the most interesting features of GSI, and are possible thanks to something called *proxy certificates*. Before looking into these concepts in detail, let's first take a look at the problem they solve.

The problem

When introducing GSI, an interesting scenario was described, as shown in Figure 10-3

Figure 10-3. A scenario where delegation is necessary



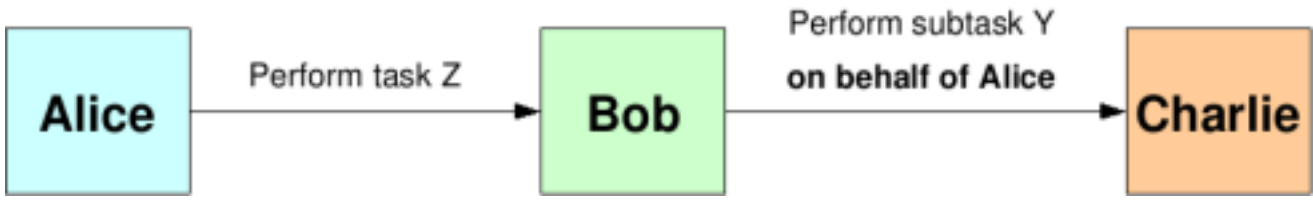
AliceOrg asks BobOrg to perform a task. Since BobOrg trusts AliceOrg, it accepts to perform the task. But let's suppose that task Z is very complex, and that one of its subtasks (Y) must be performed by a third organization: CharlieOrg. In this case, BobOrg will ask CharlieOrg to perform subtask Y but, alas!, CharlieOrg only trusts AliceOrg. What should CharlieOrg do? It has two options:

- **Turn down BobOrg's request**. CharlieOrg doesn't trust BobOrg, and that's that.

- **Accept BobOrg's request**. The 'original' requester is AliceOrg so, although CharlieOrg is answering a request from BobOrg, it will actually be carrying out a job for AliceOrg.

In this situation, it seems logical that CharlieOrg should *accept* BobOrg's request. However, CharlieOrg has to know that BobOrg's request is performed on behalf of AliceOrg, as shown in Figure 10-4.

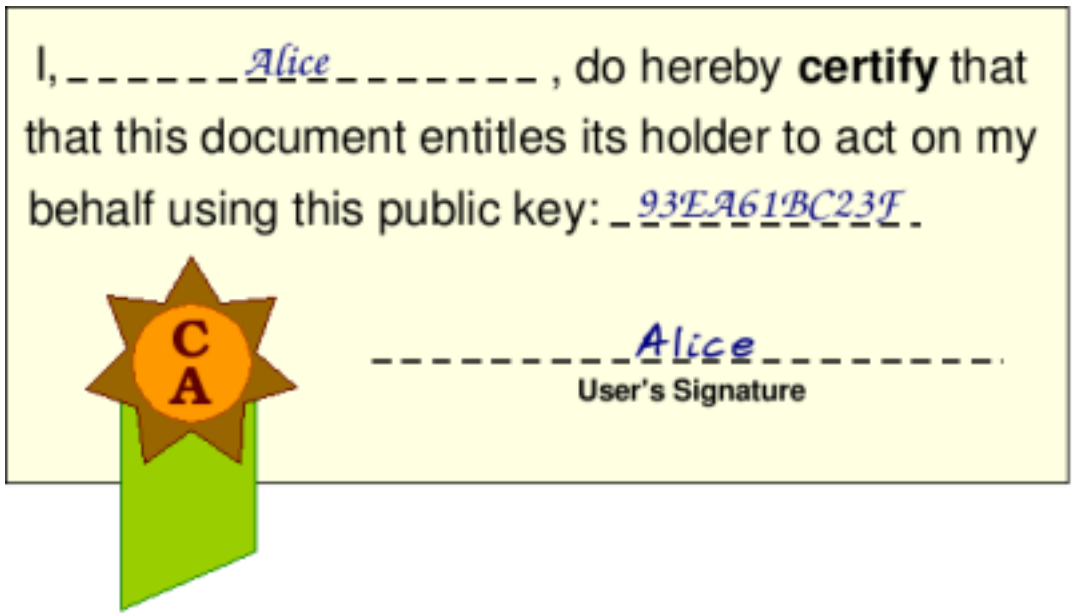
Figure 10-4. Delegation



Of course, this is not a very secure solution, since *anyone* could claim to be acting on AliceOrg's behalf! One possible solution would be for CharlieOrg to contact AliceOrg every time it receives a request on AliceOrg's behalf. However, this could be a bit of a nuisance. Imagine that task Z is composed of 20 different subtasks, and that each subtask is dispatched to a different organization by BobOrg. AliceOrg would be flooded with messages saying "BobOrg just asked me to perform a task on your behalf... can you confirm that this is correct?". In response, AliceOrg would have to authenticate itself with all those organizations and give a confirmation.

A more elegant solution would be to somehow make CharlieOrg believe that BobOrg *is* AliceOrg. In other words, it would interesting to find a legitimate way for BobOrg to demonstrate that it is, in fact, acting on AliceOrg's behalf. One way of doing this would be for AliceOrg to 'lend' its public and private key pair to BobOrg. However, this is absolutely out of the question. Remember, the private key has to remain *secret*, and sending it to another organization (no matter how much you trust them) is a *big* breach in security. What we really need is a special type of certificate like the one shown in Figure 10-5.

Figure 10-5. A proxy certificate



The solution: proxy certificates

The certificate shown in Figure 10-5 is a *proxy certificate*. Webster's Dictionary defines 'proxy' as "The instrument by which a person is empowered to transact the affairs of another". As you can see in the picture, the proxy certificate allows the holder of the certificate to act on A's behalf. In fact, it's very similar to the X.509 digital certificates seen in , except that it's not signed by a Certificate Authority; it's signed by an end user. We can be sure that the certificate is authentic by checking its signature (Organization A digitally signs the certificate, as described in).

But, what about the proxy certificate's public key? Whose public key is it? Organization A's? Organization B'? The answer is 'neither'. A proxy certificate has a private-public key pair generated specifically for the proxy certificate. This private-public key pair is mutually agreed upon by both parties (in this case, A and B), and Organization A will only allow the holder of *that* private-public key pair to act on its behalf (in this case, B). The exact mechanism by which the proxy certificate is generated by A and B will be explained later on.

There is, however, something missing from the picture. Allowing someone to act *unconditionally* on your behalf is a risky affair. Sure, you might trust them now, for the particular task you want to do, but someone from Organization B might use the proxy certificate in the future to carry out some mischievous deeds on your behalf. Therefore, the lifetime of the certificate is usually very limited (for example, to 12 hours). This means that, if the proxy certificate is compromised, the attacker won't be able to make much use of it. Furthermore, proxy certificates extend ordinary X.509 certificates with extra security features to limit their functionality even more (for example, by specifying that a proxy certificate can only be used for certain tasks). Summing up, a more correct representation of a proxy certificate would be the one shown in Figure 10-6.

Figure 10-6. A proxy certificate with a limited lifetime

What the solution achieves: Delegation and single sign-on (and more)

A proxy certificate allows a user to act on another user's behalf. This is more properly called *credential delegation*, since proxy certificates allow a user to effectively *delegate* a set of credentials (the user's identity) to another user. This solves the problem originally posed, since B could use a proxy certificate (signed by A, of course) to prove that it is acting on A's behalf. Organization C would then accept B's request.

By using proxy certificates we also get another desirable feature: *single sign-on*. Without proxy certificates, Organization A would have to authenticate itself with all the organizations that receive requests 'on behalf of A'. In practice, this mean that the user in Organization A with permission to read the private key would have to access the key each time a mutual authentication is needed. Since private keys are usually protected by a password, this means that the user would have to *sign on* (provide the password) to access the key and perform authentication. Using proxy certificates, the user only has to sign in *once* to create the proxy certificate. The proxy certificate is then used for all subsequent authentications.

Finally, although we've centered on the advantages of proxy certificates for delegation, these certificates have other features that make them interesting for other purposes. For example, they can be used locally: generating a proxy certificate that authorizes myself to act on my behalf. This might sound silly, but is actually very useful since I can use the proxy certificate for all my secure conversations, instead of using my public-private key pair directly. This reduces the risk of having my conversations compromised

because an attacker would only have a chance to crack the proxy's key pair, and not my personal one (which would only be used to generate the proxy certificate).

The specifics

At this point, you might be truly impressed at how masterfully proxy certificates allow us to delegate credentials in a completely secure manner. Then again, maybe not :-) If you are not willing to take a leap of faith when we say "Proxy certificates are really nifty!", and are not totally convinced that they are secure, this section gives a much more detailed look at the process of creation and validation of a proxy certificate. You can safely skip it unless you really really really need a more detailed explanation.

How a proxy certificate is generated

We've said that a proxy certificate can be used to delegate a user's credentials to another, different user. How is this achieved in a secure manner? For example, let's suppose that (as shown in Figure 10-3) B needs A's credentials so it can make a request to C. B, therefore, needs a proxy certificate signed by A. Let's take a close look at the process used to generate that certificate.

1. B generates a public/private key pair for the proxy certificate.
2. B uses the key pair to generate a certificate request, which will be sent to A using a secure channel. This certificate request includes the proxy's public key, but *not* the private key.
3. Supposing A agrees to delegate its credentials to B, Organization A will use its private key to digitally sign the certificate request.
4. A sends the signed certificate back to B using a secure channel.
5. B can now use the proxy certificate to act on A's behalf.

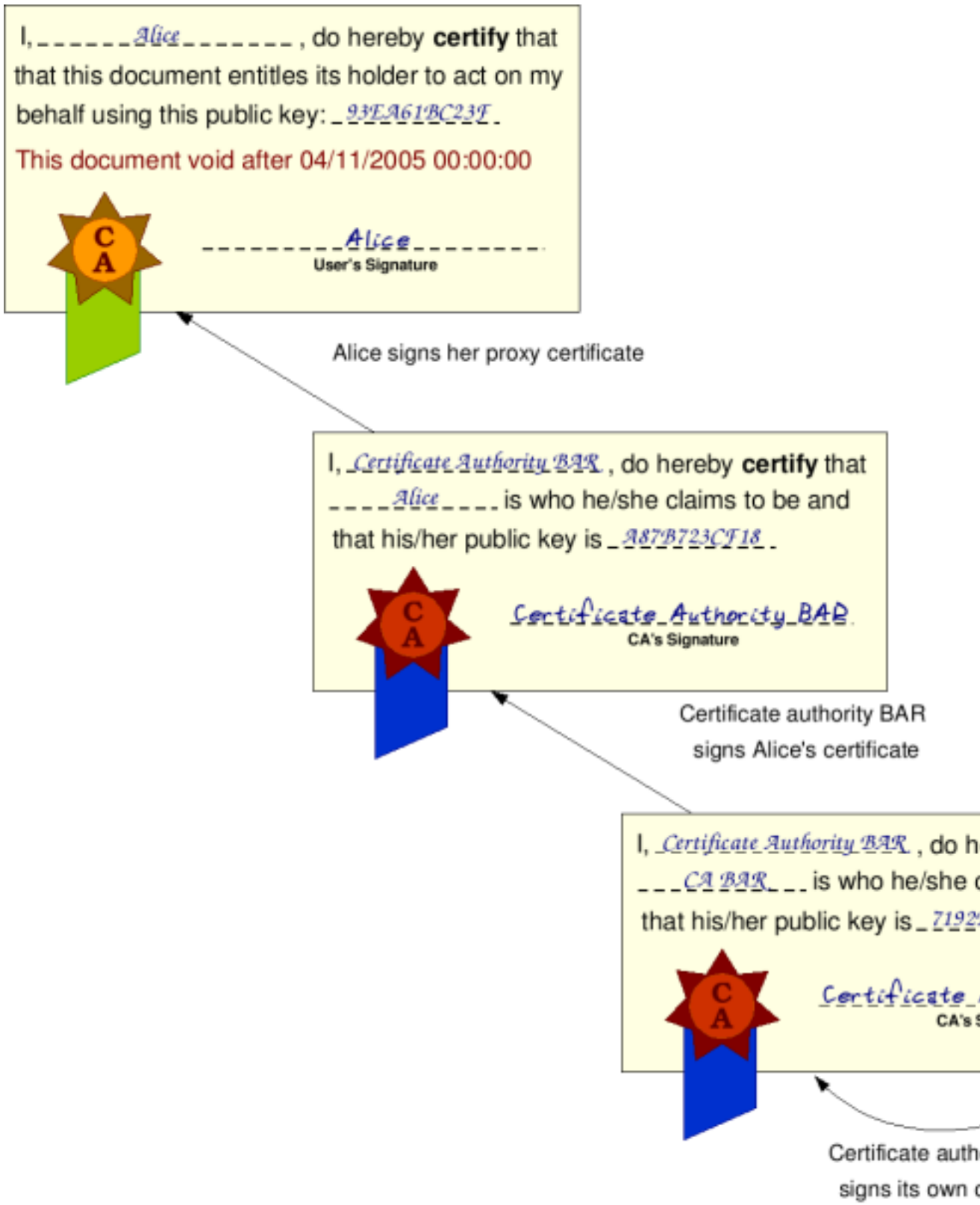
Notice how the proxy's private key is never transmitted between A and B. This is also true of A's private key.

Validation of a proxy certificate

Now let's take a look at C. When B sends a request 'on behalf of A', and sends C the proxy certificate, how can C validate the proxy certificate? In other words, how can C be absolutely sure that B *is* acting on A's behalf?

The process of validating a proxy certificate is practically identical to the process of validating an ordinary certificate, as described in . The main difference is that the proxy certificate is not signed by a Certificate Authority, it's signed by a user. In our example, the proxy certificate is signed by A, which means that we need A's public key to test its authenticity. Since C is unlikely to have A's certificate, a request that uses a proxy certificate generally also sends the delegator's certificate, so the proxy certificate can be validated. Since the delegator's certificate will be signed by a Certificate Authority, the only step left is to validate the Certificate Authority's signature. Figure 10-7 shows the chain of signatures that we could find in a proxy certificate.

Figure 10-7. Validation of a proxy certificate



More on proxy certificates

There's a lot more to proxy certificates than what has been explained in this chapter. For example, you can use proxy certificates to sign other proxy certificates. However, for the purposes of this tutorial , the material covered here should be enough. If you want to take a closer look at proxy certificates, and everything that can be done with them, we highly recommend reading RFC 3820, Internet X.509 Public Key Infrastructure Proxy Certificate Profile, available at http://www.ietf.org/rfc/rfc3820.txt.

Container, service, and resource security

Finally, it should be noted that many of the features described in this chapter can be specified at three levels: container, service, and resource level. Of special interest is the fact that we can configure security at the resource level. For example, we can set different authorization mechanisms for a service and its resources, so that stateless operations can be performed without authorization, but stateful operations do require an authorized user.

Throughout the following chapters, we will see exactly what can be configured at each of the three levels.

Chapter 11. Writing a Secure Math Service

In this chapter we will add security to our MathService example. We will see that, once we have configured GSI, adding basic security to a service is very simple. The example in this chapter will highlight the changes necessary to make a service secure. In particular, the code will be similar to the singleton example seen in . In the following chapters, we will continue to build on this example as we take a closer look at more elaborate security scenarios.

> **Note:** This chapter assumes that you have set up security in GT4 as explained in the official installation guide (http://globus.org/toolkit/docs/4.0/admin/docbook/). In particular, to work through all the following examples, you will need to make sure you have two separate users: a special `globus` account and a normal user account which we will call `globus4user` (this can be your normal user account). The user account must have a valid digital certificate.

A secure service

The service interface

Adding security to a service does *not* affect the service interface. However, for the purposes of this example, and the following examples, we will be using a new MathService interface with 4 operations (`add`, `subtract`, `multiply`, and `divide`). We are simply doing this because, further on, it will allow us to configure each operation with a different security configuration (and four simply happens to be a convenient number of operations).

> **Note:** The WSDL file for this example can be found here:
> `$EXAMPLES_DIR/schema/examples/MathService_instance_4op/Math.wsdl`

The service implementation

At this point, we don't have to modify the service implementation either, since we will be able to add security simply by modifying the WSDD file. However, we *will* be adding a private method `logSecurityInfo` to the service class to print out certain security information.

> **Note:** The code for the service can be found in
> `$EXAMPLES_DIR/org/globus/examples/services/security/first/impl/MathService.java`
>
> The code for the resource can be found in
> `$EXAMPLES_DIR/org/globus/examples/services/security/first/impl/MathResource.java`
>
> The code for the resource home can be found in
> `$EXAMPLES_DIR/org/globus/examples/services/security/first/impl/MathResourceHome.java`

First, let's take a look at the `logSecurityInfo` method. This method will print out a lot of security information. At this point, we are only interested in a snippet of code that prints out the client's identity. This will allow us to verify that authentication is taking place and that the service correctly receives the client's credentials. In the following chapters, we will see what the rest of `logSecurityInfo` prints out, and what that information means.

```
private void logSecurityInfo(String methodName)
{
Subject subject;
logger.info("SECURITY INFO FOR METHOD '" + methodName + "'");

// Print out the caller
String identity = SecurityManager.getManager().getCaller();
logger.info("The caller is:" + identity);

// Print out more security information
}
```

Next, the implementation of the remote operations is exactly the same as in a non-secure service. The only difference is that we will be calling the `logSecurityInfo` method in each of them. For example, the `add` method looks like this:

```
public AddResponse add(int a) throws RemoteException {
logSecurityInfo("add");

MathResource mathResource = getResource();
mathResource.setValue(mathResource.getValue() + a);
mathResource.setLastOp("ADDITION");

return new AddResponse();
}
```

Finally, remember that, strictly speaking, we are not modifying the Java files at all. We are simply adding some logging code to keep track of what's happening in the service. At this point, adding security will affect only the deployment files. Later on, more complicated security scenarios will require that we modify the service implementation.

The security descriptor

The heart of a secure service is its *security descriptor*. This file specifies the security configuration for a service. One of the really neat things about the security descriptor is that it centralizes practically all the security configuration for a service. So, if we decide to modify some security aspects of a service, we will only need to modify the security descriptor, *not* the Java files.

In the next chapter, we will take a much closer look at this special file and its syntax. For now, we will be using the following security descriptor:

```
<securityConfig xmlns="http://www.globus.org">
```

```
<authz value="none"/>

</securityConfig>
```

> **Note:** This is file `$EXAMPLES_DIR/org/globus/examples/services/security/first/etc/security-config-first.xml`

This security descriptor simply specifies that we will not be performing any authorization (`none`). As we will see in the next chapter, the fact that we have not specified anything else basically means that the client will be free to use any type of security it wants. For example, we will be configuring our client to use GSI Secure Conversation.

Of course, we'll need to tell our service that we want it to use that security descriptor. To do this, we have to add the following parameter to the WSDD file. Notice that the path to the security descriptor is relative to `$GLOBUS_LOCATION`.

```
<parameter name="securityDescriptor"
value="etc/org_globus_examples_services_security_first/security-config-first.xml"/>
```

> **Note:** The WSDD file for this service is `$EXAMPLES_DIR/org/globus/examples/services/security/first/deploy.wsdd`
>
> Our service's name is `"examples/security/first/MathService"`.

A secure client

The client used to invoke the secure service will be almost identical to all the clients seen so far. The only difference is that we will be instructing the client to use GSI Secure Conversation with encryption and no client-side authorization. Believe it or not, this requires two simple lines of code:

```
❶
((Stub)math)._setProperty(Constants.GSI_SEC_CONV,Constants.ENCRYPTION);
❷
((Stub)math)._setProperty(Constants.AUTHORIZATION,NoAuthorization.getInstance());
```

❶ We're telling the stub to use GSI Secure Conversation with encryption.

❷ We're telling the stub to use no *client-side* authorization. Remember that there is a difference in GSI between client-side and server-side authorization. Take a look at the .

Besides those two lines, the rest of the client is practically identical to the ones we've already seen. The only difference is that we will be putting the calls to the remote operations inside `try...catch` blocks to observe how certain exceptions are raised in certain circumstances (we'll see this in the following chapters).

```
package org.globus.examples.clients.MathService_instance_4op;

import javax.xml.rpc.Stub;

import org.apache.axis.message.addressing.Address;
import org.apache.axis.message.addressing.EndpointReferenceType;

import org.globus.axis.util.Util;
import org.globus.examples.services.security.first.impl.MathQNames;
import org.globus.examples.stubs.MathService_instance_4op.MathPortType;
import org.globus.examples.stubs.MathService_instance_4op.service.MathServiceAddressingLoca
import org.globus.wsrf.impl.security.authorization.NoAuthorization;
import org.globus.wsrf.security.Constants;
import org.oasis.wsrf.properties.GetResourcePropertyResponse;

public class Client_GSISecConv_Encrypt {

public static void main(String[] args) {
MathServiceAddressingLocator locator = new MathServiceAddressingLocator();
GetResourcePropertyResponse valueRP;
String value;

try {
String serviceURI = args[0];

// Create endpoint reference to service
EndpointReferenceType endpoint = new EndpointReferenceType();
endpoint.setAddress(new Address(serviceURI));
MathPortType math = locator.getMathPortTypePort(endpoint);

// Get PortType
math = locator.getMathPortTypePort(endpoint);

// Setup security options
((Stub)math)._setProperty(Constants.GSI_SEC_CONV,Constants.ENCRYPTION);
((Stub)math)._setProperty(Constants.AUTHORIZATION,NoAuthorization.getInstance());

// Perform an addition
try {
math.add(60);
System.out.println("Addition was successful");
} catch (Exception e) {
System.out.println("[add]      ERROR: " + e.getMessage());
}

/* Similar calls to subtract(), multiply(), divide(), and getResourceProperty */

} catch (Exception e) {
e.printStackTrace();
}
}
}
```

Note: This file is
`$EXAMPLES_DIR/org/globus/examples/clients/MathService_instance_4op/Client_GSISecConv_Encrypt.java`

Trying it out

We are now ready to give this secure service a try.

Compile and deploy

First of all, we'll need to build the service:

```
./globus-build-service.sh sec_first
```

Now, we have to deploy it. Remember that you have to do this from the `globus` account:

```
globus-deploy-gar $EXAMPLES_DIR/org_globus_examples_services_security_first.gar
```

Starting the container

At this point you might be thinking that we will now be running the container *without* the `-nosec` flag we've been using so far to "deactivate security". Well, you thought wrong! :-)

```
globus-start-container -nosec
```

At this point, we can clarify what the `-nosec` flag does. It only deactivates *transport-level* security, but not message-level security. So, we can still use GSI Secure Conversation and GSI Secure Message. The reason why we're not using transport-level security (yet) is because, as part of our test of this service, we will be using a tool included with the Globus Toolkit that can intercept the SOAP messages. However, this tool won't work if we use transport-level security, so you should use the `-nosec` flag if you want to participate in our little experiment. In the following chapters, on the other hand, you can use the `-nosec` flag at your discretion (unless otherwise noted).

Compiling the client

Let's compile the client:

```
javac \
-classpath ./build/stubs/classes/:$CLASSPATH \
org/globus/examples/clients/MathService_instance_4op/Client_GSISecConv_Encrypt.java
```

Running the client

Before running the client, we will need to create a proxy certificate for our user account. We have to do this because the default behavior in the client-side is to use a proxy certificate for authentication. In the next chapter we will see how we can configure a client to use a specific set of credentials, instead of using a proxy certificate.

To create a proxy certificate, run the following from your user account:

```
grid-proxy-init
```

You will see the following:

```
Your identity: /O=Globus/OU=GT4 Examples/CN=Globus 4 User
Enter GRID pass phrase for this identity:
```

The password you must enter is the one you entered when creating your user certificate (as described in the official installation guide (http://globus.org/toolkit/docs/4.0/admin/docbook/)). Once you've entered the password, you will see the following:

```
Creating proxy ..................................... Done
Your proxy is valid until: Sun Apr 24 04:28:26 2005
```

> **Warning**
>
> Globus proxy certificates expire by default in 12 hours. If you get a "proxy expired" or "no valid credentials found" error message later on, this probably means that your proxy certificate has expired. Simply create a new one using `grid-proxy-init`.

Now, run the client:

```
java \
-classpath ./build/stubs/classes/:$CLASSPATH \
org.globus.examples.clients.MathService_instance_4op.Client_GSISecConv_Encrypt \
http://127.0.0.1:8080/wsrf/services/examples/security/first/MathService
```

If all goes well, you should see this in the client side:

```
Addition was successful
Subtraction was successful
Multiplication was successful
Division was successful
Current value: 20
```

And the following on the server side:

```
SECURITY INFO FOR METHOD 'add'
The caller is: /O=Globus/OU=GT4 Examples/CN=Globus 4 User

... other security information ...
```

> **Note:** Remember that the `logSecurityInfo` will also print out a lot of other information. Don't worry about that information right now. It will be explained later on.

Notice how the service has correctly authenticated the client, and prints out its distinguished name: `/O=Globus/OU=GT4 Examples/CN=Globus 4 User`.

Does this really work?

After all the work we've gone through to setup security, you might be a bit disappointed. After all, we've gone through all the trouble of setting up a CA and some certificates to end up writing a MathService client that behaves just like all the other MathService clients we've already seen in the tutorial. Ho hum. You're probably asking yourself: "Yeah, but is this really doing all that encryption thingy?"

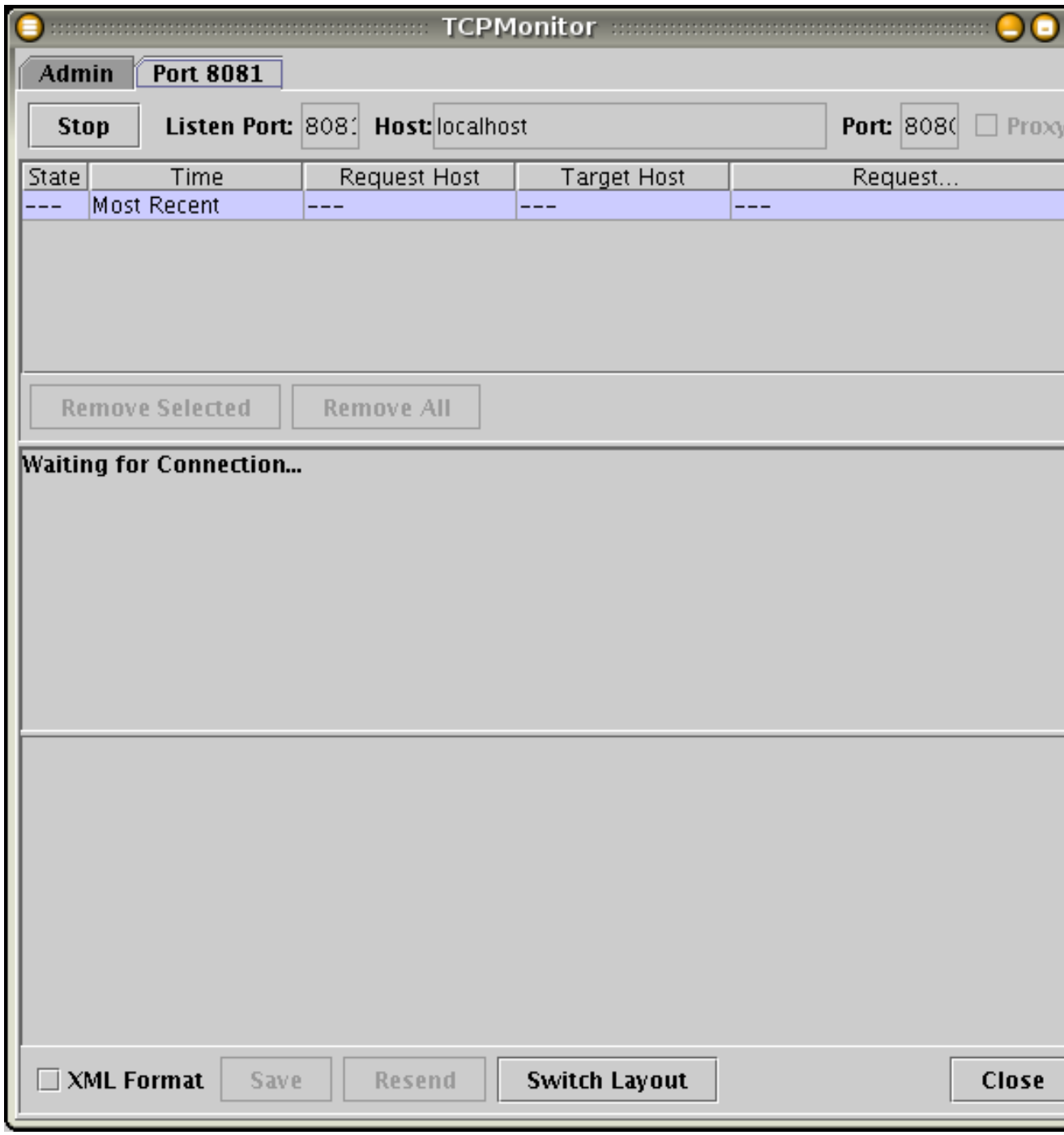
To empirically prove that it is doing the 'encryption thingy', we are going to use an Apache Axis tool called TCPMonitor that is included with the toolkit. This tool allows us to intercept the data that is sent from the client to the server (and vice versa). We will see how the information is, in fact, encrypted.

To start TCPMonitor, run this:

```
java org.apache.axis.utils.tcpmon 8081 localhost 8080
```

This starts an instance of the TCPMonitor (Figure 11-1). What the monitor will do is listen on port 8081 and redirect all the traffic it receives on that port to port 8080 (which is where our container is listening). This means that TCPMonitor acts like a proxy, not like a sniffer, so we'll have to tell our client to make the invocation on port 8081 to be able to see what kind of data is being sent.

Figure 11-1. TCPMonitor interface (1)



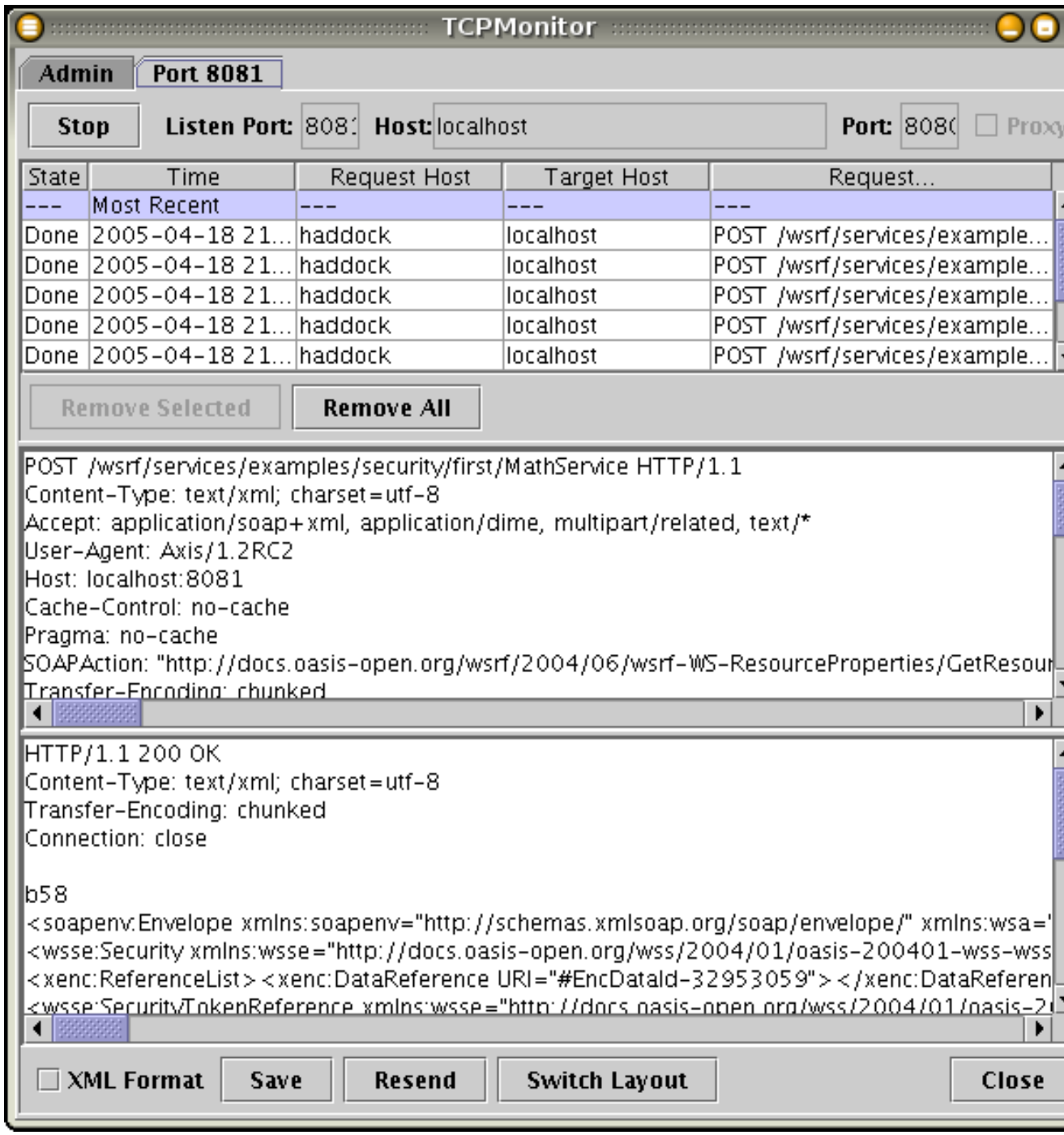
Note: Make sure you check the "XML Format" box in TCPMonitor. This will make reading the messages much easier.

Let’s run the client again. Make sure to change ’8080’ for ’8081’ so that the invocation will go through the TCPMonitor. Otherwise, we won’t be able to see it.

```
java \
-classpath ./build/stubs/classes/:$CLASSPATH \
org.globus.examples.clients.MathService_instance_4op.Client_GSISecConv_Encrypt \
http://127.0.0.1:8081/wsrf/services/examples/security/first/MathService
```

Once you’ve invoked the service, the TCPMonitor will reflect that it has intercepted some connections on port 8081 (Figure 11-2). The top list shows a list of the connections. You can select any of them to see what was sent to the server (top text area) and what the server replied to the client (bottom text area).

Figure 11-2. TCPMonitor interface (2)



Five calls and eight connections... huh?: TCPMonitor should show 8 connections. This might seem a bit odd considering that our client only makes *five* calls to MathService (`add`, `subtract`, `multiply`, `divide`, and `getResourceProperty`). However, remember that we're using GSI Secure Conversation.

As described in the previous chapter, GSI Secure Conversation involves the creation of a *security context* before the client and server can actually communicate. This accounts for the first three connections. The client and server are agreeing on the details of the secure context. Once the context is created, the calls can proceed as normal.

Let's take a look at the fourth connection, corresponding to the *add* invocation:

```
<soapenv:Envelope xmlns:soapenv="http://schemas.xmlsoap.org/soap/envelope/"
  xmlns:wsa="http://schemas.xmlsoap.org/ws/2004/03/addressing" xmlns:xenc="http://www.w3.or
  xmlns:xsd="http://www.w3.org/2001/XMLSchema" xmlns:xsi="http://www.w3.org/2001/XMLSchema-
   <soapenv:Header>
      <wsse:Security
        xmlns:wsse="http://docs.oasis-open.org/wss/2004/01/oasis-200401-wss-wssecurity-sece
        soapenv:mustUnderstand="1">
         <wsc:SecurityContextToken xmlns:wsc="http://schemas.xmlsoap.org/ws/2004/04/sc"
           xmlns:wsu="http://docs.oasis-open.org/wss/2004/01/oasis-200401-wss-wssecurity-ut
           wsu:Id="SecurityContextToken-15021407">
            <wsc:Identifier>8439eb60-b079-11d9-bf45-98ad001497de</wsc:Identifier>
         </wsc:SecurityContextToken>
         <xenc:ReferenceList>
            <xenc:DataReference URI="#EncDataId-749304"></xenc:DataReference>
         </xenc:ReferenceList>
      </wsse:Security>
      <wsa:MessageID soapenv:mustUnderstand="0">uuid:837bf290-b079-11d9-a882-a1d8507fabe7</
      <wsa:To soapenv:mustUnderstand="0">
        http://127.0.0.1:8081/wsrf/services/examples/security/first/MathService
      </wsa:To>
      <wsa:Action soapenv:mustUnderstand="0">
        http://www.globus.org/namespaces/examples/core/MathService_instance_4op/MathPortTyp
      </wsa:Action>
      <wsa:From soapenv:mustUnderstand="0">
         <wsa:Address>http://schemas.xmlsoap.org/ws/2004/03/addressing/role/anonymous</wsa:
      </wsa:From>
   </soapenv:Header>

   <soapenv:Body>
      <xenc:EncryptedData Id="EncDataId-749304" Type="http://www.w3.org/2001/04/xmlenc#Cont
         <xenc:EncryptionMethod Algorithm="http://www.globus.org/2002/04/xmlenc#gssapi-enc"
         <ds:KeyInfo xmlns:ds="http://www.w3.org/2000/09/xmldsig#">
            <wsse:SecurityTokenReference
              xmlns:wsse="http://docs.oasis-open.org/wss/2004/01/oasis-200401-wss-wssecurit
               <wsse:Reference URI="#SecurityContextToken-15021407"></wsse:Reference>
            </wsse:SecurityTokenReference>
         </ds:KeyInfo>
         <xenc:CipherData>
            <xenc:CipherValue>
            FwMAAYCNYHoPTT6WgG096yj0NPONpNXp3u8tVPbPBjZzqzFtx03Yh1tMEI6ebObR1Rvr8ihBk+EtBsC
            zacALBaLpie0A1BfO8QjjDNGvv6KsDrydY2qywSuaZvqclUus2eAnPjW3ewanPqemntYfSHKW0X81wk
            Yg/R4osS2PR+/aqAwAb8vdZtzDdBkpyIDfCsMrQnZA1sarFbdZPT7buNIPRJ8vja3/icV5yB4WZg8so
            gznsJj0pf67BcowXO+swp0uSAnjczNEyDRafk6HaIPeeMpZWLGNL11BrfHM1pDGs0foDCvbFApfFJGz
            0Bp8gc0hlB+hrsc6Wyf8RiQ/kxBfMYZ3hKY8fah4oRaq39/sJvA8pXnfxvp1EKonQKhOh/yYYGkuzMI
```

```
            UdAF1zwwqyPK4zYBHsz1Qe0I8lD2OmTjQw=
            </xenc:CipherValue>
         </xenc:CipherData>
      </xenc:EncryptedData>
   </soapenv:Body>

</soapenv:Envelope>
```

Holy gibberish, Batman! :-)

Notice how only certain parts of the message are encrypted, not the *whole* message. Remember that this is because GSI Secure Conversation is a form of *message-level* security (explained in). This means that we only encrypt the contents of the message, but not the whole message. Notice how the message still reveals what the service URI is, along with the operation name, but the actual call itself is encrypted. If we wanted the whole message to be encrypted we would need to use *transport-level* security (GSI Transport).

IV. GT4 Information Services
[Coming soon]

Work in progress!

Although the tutorial will eventually include chapters on GT4 Information Services, work on these chapters hasn't begun yet. If you need information on this subject, check out the official documentation (http://www.globus.org/toolkit/docs/4.0/info/).

If you want to know when these chapters will see the light of day, check out the TODO page (http://gdp.globus.org/gt4-tutorial/TODO.html), which includes a roadmap of future versions. Also, if you'd like to help out with the development of these chapters, please don't hesitate to contact me (http://gdp.globus.org/gt4-tutorial/contact.html).

V. GT4 Execution Management
[Coming later]

Work in progress!

Although the tutorial will eventually include chapters on GT4 Execution Management, work on these chapters hasn't begun yet. If you need information on this subject, check out the official documentation (http://www.globus.org/toolkit/docs/4.0/execution/).

If you want to know when these chapters will see the light of day, check out the TODO page (http://gdp.globus.org/gt4-tutorial/TODO.html), which includes a roadmap of future versions. Also, if you'd like to help out with the development of these chapters, please don't hesitate to contact me (http://gdp.globus.org/gt4-tutorial/contact.html).

VI. GT4 Data Management [Coming even later]

Work in progress!

Although the tutorial will eventually include chapters on GT4 Data Management, work on these chapters hasn't begun yet. If you need information on this subject, check out the official documentation (http://www.globus.org/toolkit/docs/4.0/data/).

If you want to know when these chapters will see the light of day, check out the TODO page (http://gdp.globus.org/gt4-tutorial/TODO.html), which includes a roadmap of future versions. Also, if you'd like to help out with the development of these chapters, please don't hesitate to contact me (http://gdp.globus.org/gt4-tutorial/contact.html).

VII. Appendices

Appendix A. How to...

...write a WSDL description of your WSRF stateful Web service

This HOWTO shows you how to write a simple WSDL description of a portType in a step-by-step fashion. Although it should be easy to follow those same steps to create other simple WSDL files, this is not meant as an exhaustive WSDL guide. Anyone seeking to write more complex portTypes (for example, passing complex classes instead of primitive types -int. string, etc.- as parameters or return values) should definitely consider learning WSDL and XML Schema. A couple of references are provided at the end of the appendix in case you want to learn more.

That said, let's start writing WSDL! We are going to write the WSDL description corresponding to the following Java interface:

```
public interface Math
{
  public void add(int a);

  public void subtract(int a);

  public int getValueRP();
}
```

Furthermore, our portType will have two resource properties: "value" of type integer and "lastOp" of type string. This is the interface used in many of the tutorial's examples.

The bare bones of our WSDL file

First of all, we have to write the root element of the WSDL file, which is <definitions>.

```
<?xml version="1.0" encoding="UTF-8"?>
<definitions name="MathService"
    targetNamespace="http://www.globus.org/namespaces/examples/core/MathService_instance"
    xmlns="http://schemas.xmlsoap.org/wsdl/"
    xmlns:tns="http://www.globus.org/namespaces/examples/core/MathService_instance"
    xmlns:wsdl="http://schemas.xmlsoap.org/wsdl/"
    xmlns:wsrp="http://docs.oasis-open.org/wsrf/2004/06/wsrf-WS-ResourceProperties-1.2-draf
    xmlns:wsrpw="http://docs.oasis-open.org/wsrf/2004/06/wsrf-WS-ResourceProperties-1.2-dra
    xmlns:wsdlpp="http://www.globus.org/namespaces/2004/10/WSDLPreprocessor"
    xmlns:xsd="http://www.w3.org/2001/XMLSchema">

</definitions>
```

This tag has two important attributes:

- **name**: The 'name' of the WSDL file. Not related with the name of the portType.

- **targetNamespace**: The target namespace of the WSDL file. This means that all the PortTypes and operations defined in this WSDL file will belong to this namespace.

> **XML Namespaces:** XML namespaces are basically a way of grouping similar 'things' together. We're using the somewhat vague term 'things' because XML Namespaces are used not only in WSDL, but in many XML languages, so just about anything can be grouped into an XML Namespace (not only portTypes and operations, which are specific to WSDL). The XML Namespace has to be a valid URI, but it doesn't necessarily have to be real (in fact, if you try to point your web browser to the URI we're using, you'll get a Page Not Found error).

The root element is also used to declare all the namespaces we are going to use. Notice how the `tns` namespace is the Target NameSpace. The rest of the namespace declarations should be copied verbatim.

Next up, we need to import a WSDL file with definitions we'll need later on.

```
<wsdl:import
    namespace=
    "http://docs.oasis-open.org/wsrf/2004/06/wsrf-WS-ResourceProperties-1.2-draft-01.wsdl"
    location="../../wsrf/properties/WS-ResourceProperties.wsdl" />
```

> **Note:** In general, we will import the WSDL file of every WSRF specification (such as WS-ResourceProperties) we intend to use in our service. As we'll see later on, we will need to import other WSDL files as we start looking at other parts of the WSRF specification.

The Port Type

Now we're going to define our portType, using the `<portType>` tag:

```
<?xml version="1.0" encoding="UTF-8"?>
<definitions ... >

<portType name="MathPortType"
    wsdlpp:extends="wsrpw:GetResourceProperty"
    wsrp:ResourceProperties="tns:MathResourceProperties">

<operation name="add">
<input message="tns:AddInputMessage"/>
<output message="tns:AddOutputMessage"/>
</operation>

<operation name="subtract">
<input message="tns:SubtractInputMessage"/>
<output message="tns:SubtractOutputMessage"/>
</operation>
```

```
<operation name="getValueRP">
<input message="tns:GetValueRPInputMessage"/>
<output message="tns:GetValueRPOutputMessage"/>
</operation>

</portType>

</definitions>
```

The <portType> tag has three important attributes:

- **name**: Name of the PortType.
- **wsdlpp:extends**: This is not a standard part of WSDL, but part of the `WSDLPreprocessor` namespace provided by Globus (notice how we declared `wsdlpp` in the namespace declarations). If we were writing strict WSDL, the only way of including operations and portTypes from WSRF specifications would be to actually copy and paste those definitions from the spec's WSDL file into our own WSDL file. This, of course, is very error-prone. For our convenience, Globus provides a WSDL Preprocessor that does that automatically for us. By using the `wsdlpp:extends` attribute, we are telling the WSDL Preprocessor to include the `GetResourceProperty` portType from the WS-ResourceProperties WSDL file. This portType is first used in .
- **wsrp:ResourceProperties**: This attribute specifies what the service's resource properties are. The meaning of this attribute is explained further on.

> **Warning**
>
> Make sure you read the warning regarding the WSDL Preprocessor in .

Inside the `<portType>` we have an `<operation>` tag for the three operations exposed in our web service: `add`, `subtract`, and `getValueRP`. They are all very similar, so let's just take a closer look at `add`'s `<operation>` tag:

```
<operation name="add">
  <input message="tns:AddInputMessage"/>
  <output message="tns:AddOutputMessage"/>
</operation>
```

The `<operation>` tag has an `<input>` tag and an `<output>`. These two tags have a `message` attribute, which specifies what message should be passed along when the operation is invoked (input message) and when it returns successfully (output message). So, we'll need to define the messages of our operations.

The messages

The following are the messages for the `add` operation. The messages for the `subtract` and `getValueRP` operations are identical.

```
<?xml version="1.0" encoding="UTF-8"?>
```

```
<definitions ... >

<message name="AddInputMessage">
<part name="parameters" element="tns:add"/>
</message>
<message name="AddOutputMessage">
<part name="parameters" element="tns:addResponse"/>
</message>

<!-- PortType -->

</definitions>
```

Notice how the name of each message has to be the same as the one written in the message attribute of the `<input>` and `<output>` tags. However, it turns out messages are composed of `<part>`s! Our messages will only have one part, in which a single XML element is passed along. For example, the `AddOutputMessage` will contain the addResponse element (notice how it is part of the `tns` namespace, the target namespace).

The response and request types

The definition of these elements is done using XML Schema inside a new tag: the `<types>` tag. The following would be the definition of the `add` and `addResponse` elements:

```
<?xml version="1.0" encoding="UTF-8"?>
<definitions ... >

<types>
<xsd:schema targetNamespace="http://www.globus.org/namespaces/examples/core/MathService_ins
    xmlns:tns="http://www.globus.org/namespaces/examples/core/MathService_instance"
    xmlns:xsd="http://www.w3.org/2001/XMLSchema">


<!-- REQUESTS AND RESPONSES -->

<xsd:element name="add" type="xsd:int"/>
<xsd:element name="addResponse">
<xsd:complexType/>
</xsd:element>

<!-- more type definitions -->

</xsd:schema>
</types>

<!-- Messages -->

<!-- PortType -->

</definitions>
```

The `<types>` tag contains an `<xsd:schema>` tag. The attributes of the `<xsd:schema>` should be copied verbatim, except the `targetNamespace`, which should be the same as the target namespace of the WSDL document.

The `add` element (which, remember, is part of the *input* message of the `add` operation) represents the single input parameter of our add operation, and thus has an attribute specifying its type (notice how the type attribute is equal to `xsd:int`, the integer type in XML Schema).

If we had wanted our add operation to receive two parameters, we would have to declare the `add` element like this:

```
<xsd:element name="add">
    <xsd:complexType>
    <xsd:sequence>
        <xsd:element name="a1" type="xsd:int"/>
        <xsd:element name="a2" type="xsd:int"/>
    </xsd:sequence>
    </xsd:complexType>
</xsd:element>
```

In this case we're saying that the add operation has two parameters (a1 and a2) of type integer. Notice that to specify multiple parameters we need to use the XML Schema `complexType` tag.

As for the `addResponse` element (part of the output message of the add operation, i.e. the return value), it contains an empty `complexType`, since the add operation doesn't return anything.

The type definitions for the `subtract` and `getValueRP` operation are defined similarly.

Declaring the resource properties

We are very nearly done. There is only one thing left to do: declare our service's resource properties. This is also done in the `<types>` part of the WSDL document, inside the `<schema>` tag along with all the declarations we have just seen.

First of all, let's take another quick look at our portType:

```
<portType name="MathPortType"
    wsdlpp:extends="wsrpw:GetResourceProperty"
    wsrp:ResourceProperties="tns:MathResourceProperties">

<!-- operations --">

</portType>
```

The `wsrp:ResourceProperties` attribute specifies what the service's resource properties are. In our case, the resource properties are contained in an element called `MathResourceProperties` which we must declare in the `<types>` part of the WSDL document.

```
<types>
<xsd:schema targetNamespace="http://www.globus.org/namespaces/examples/core/MathService_ins
    xmlns:tns="http://www.globus.org/namespaces/examples/core/MathService_instance"
    xmlns:xsd="http://www.w3.org/2001/XMLSchema">
```

```
<!-- Requests and responses declarations-->



<xsd:element name="Value" type="xsd:int"/>
<xsd:element name="LastOp" type="xsd:string"/>

<xsd:element name="MathResourceProperties">
<xsd:complexType>
<xsd:sequence>
<xsd:element ref="tns:Value" minOccurs="1" maxOccurs="1"/>
<xsd:element ref="tns:LastOp" minOccurs="1" maxOccurs="1"/>
</xsd:sequence>
</xsd:complexType>
</xsd:element>


</xsd:schema>
</types>
```

This declaration specifies that `MathResourceProperties` contains two resource properties, `Value` and `LastOp`, each of which appear only once (we could specify array resource properties by changing the values of `maxOccurs`).

Summing up...

The whole WSDL file would be:

```
<?xml version="1.0" encoding="UTF-8"?>
<definitions name="MathService"
    targetNamespace="http://www.globus.org/namespaces/examples/core/MathService_instance"
    xmlns="http://schemas.xmlsoap.org/wsdl/"
    xmlns:tns="http://www.globus.org/namespaces/examples/core/MathService_instance"
    xmlns:wsdl="http://schemas.xmlsoap.org/wsdl/"
    xmlns:wsrp="http://docs.oasis-open.org/wsrf/2004/06/wsrf-WS-ResourceProperties-1.2-draf
    xmlns:wsrpw="http://docs.oasis-open.org/wsrf/2004/06/wsrf-WS-ResourceProperties-1.2-dra
    xmlns:wsdlpp="http://www.globus.org/namespaces/2004/10/WSDLPreprocessor"
    xmlns:xsd="http://www.w3.org/2001/XMLSchema">

<wsdl:import
    namespace=
    "http://docs.oasis-open.org/wsrf/2004/06/wsrf-WS-ResourceProperties-1.2-draft-01.wsdl"
    location="../../wsrf/properties/WS-ResourceProperties.wsdl" />


<!============================================================

                      T Y P E S
```

```
  ============================================================>
<types>
<xsd:schema targetNamespace="http://www.globus.org/namespaces/examples/core/MathService_ins
    xmlns:tns="http://www.globus.org/namespaces/examples/core/MathService_instance"
    xmlns:xsd="http://www.w3.org/2001/XMLSchema">


<!-- REQUESTS AND RESPONSES -->

<xsd:element name="add" type="xsd:int"/>
<xsd:element name="addResponse">
<xsd:complexType/>
</xsd:element>

<xsd:element name="subtract" type="xsd:int"/>
<xsd:element name="subtractResponse">
<xsd:complexType/>
</xsd:element>

<xsd:element name="getValueRP">
<xsd:complexType/>
</xsd:element>
<xsd:element name="getValueRPResponse" type="xsd:int"/>


<!-- RESOURCE PROPERTIES -->

<xsd:element name="Value" type="xsd:int"/>
<xsd:element name="LastOp" type="xsd:string"/>

<xsd:element name="MathResourceProperties">
<xsd:complexType>
<xsd:sequence>
<xsd:element ref="tns:Value" minOccurs="1" maxOccurs="1"/>
<xsd:element ref="tns:LastOp" minOccurs="1" maxOccurs="1"/>
</xsd:sequence>
</xsd:complexType>
</xsd:element>

</xsd:schema>
</types>


<!============================================================

                    M E S S A G E S

  ============================================================>
<message name="AddInputMessage">
<part name="parameters" element="tns:add"/>
</message>
<message name="AddOutputMessage">
<part name="parameters" element="tns:addResponse"/>
```

```
</message>

<message name="SubtractInputMessage">
<part name="parameters" element="tns:subtract"/>
</message>
<message name="SubtractOutputMessage">
<part name="parameters" element="tns:subtractResponse"/>
</message>

<message name="GetValueRPInputMessage">
<part name="parameters" element="tns:getValueRP"/>
</message>
<message name="GetValueRPOutputMessage">
<part name="parameters" element="tns:getValueRPResponse"/>
</message>



<!================================================================

                      P O R T T Y P E

  ================================================================>
<portType name="MathPortType"
    wsdlpp:extends="wsrpw:GetResourceProperty"
    wsrp:ResourceProperties="tns:MathResourceProperties">

<operation name="add">
<input message="tns:AddInputMessage"/>
<output message="tns:AddOutputMessage"/>
</operation>

<operation name="subtract">
<input message="tns:SubtractInputMessage"/>
<output message="tns:SubtractOutputMessage"/>
</operation>

<operation name="getValueRP">
<input message="tns:GetValueRPInputMessage"/>
<output message="tns:GetValueRPOutputMessage"/>
</operation>

</portType>

</definitions>
```

Summing up, the basic steps involved in writing a WSDL file for a WSRF web service would be the following:

1. Write the root element `<definitions>`
2. Write the `<portType>`

3. Write an input and output `<message>` for each operation in the PortType.
4. Write the `<types>`. This includes declaring the request and response elements, along with the resource properties.

As any experienced WSDL writer should be able to tell you, there are many ways of writing WSDL (ways that allow you to write more compact WSDL). However, this is the most step-by-step method, which is probably best for beginners. Furthermore, remember this is just a very brief guide on how to write very basic WSDL. You should have no trouble adding basic operations such as void multiply(int a), but more complex portTypes will require more advanced knowledge of WSDL and XML Schema.

...use the tutorial's build script

The build script used in the tutorial is a part of the Globus Service Build Tools (http://gsbt.sourceforge.net/) project. The GSBT website includes information on how you can use the build script for your own projects. Also, you can run `./globus-build-service.sh -h` to see all the different options supported by the script.

Appendix B. Tutorial directory structure

The tutorial follows a very specific directory structure which clearly separates the interface files (WSDL), the service implementation files (Java and WSDD), and the client implementation files (Java). This directory structure must be preserved if you want the examples to compile out-of-the-box with the provided Ant build file and build script. This appendix describes the directory structure used throughout the tutorial.

Brief overview

The following is a diagram that shows where the three main types of files (build, WSDL, service, and client files) are located. The details of each type of file can be found below.

```
$EXAMPLES_DIR
  |
  |-- schema/
  |    |
  |    |-- examples/             ----->  WSDL files
  |
  |-- org/
       |
       |-- globus/
            |
            |-- examples/
                 |
                 |-- services/   ----->  Service implementation files
                 |
                 |-- clients/    ----->  Client implementation files
```

Build files

All the files needed to build the examples are included in the root of `$EXAMPLES_DIR`:

- Ant build file
- Build script
- Namespace mappings file

WSDL files

The `$EXAMPLES_DIR/schema/examples/` directory contains one subdirectory for each different service interface described in the tutorial. These subdirectories contain the WSDL file and any supporting XML Schema files.

Implementation files

The implementation classes of the services in the tutorial can be found in the following package:

`org.globus.examples.services`

Inside this package there is a subpackage for each part of the tutorial (currently only core and security). Then, inside each of these subpackages there is one sub-sub-package for each example in that part of the tutorial. For example, let's take the very first example in the tutorial. Since that particular example is the "first" service in the "GT4 Core" part of the tutorial, the implementation classes are placed inside this package:

`org.globus.examples.services.`*`core.first`*

This will be the *base package* for this example. In general, the base package will have the following format:

`org.globus.examples.services.<`*`part`*`>.<`*`example`*`>`

For example, the base package for the "resource properties" example of the "GT4 Core" part is org.globus.examples.service.**core.rp** The directory corresponding to that base package would be:

`$EXAMPLES_DIR/org/globus/examples/services/<`*`part`*`>/<`*`example`*`>/`

This is the directory where we'll place all the service's files:

```
Base package directory
  |
  |-- server-deploy.wsdd   ----->  Deployment descriptor file
  |
  |-- impl/                ----->  Implementation classes
  |
  |-- config/              ----->  Security configuration files
```

Client code

The $EXAMPLES_DIR/org/globus/examples/clients/ directory contains one subdirectory with clients for each different service interface described in the tutorial. For example, the client for the service interface in $EXAMPLES_DIR/schema/examples/**MathService_instance**/ can be found in $EXAMPLES_DIR/org/globus/examples/clients/**MathService_instance**/