The Globus Toolkit 3 Programmer's Tutorial

Borja Sotomayor

The Globus Toolkit 3 Programmer's Tutorial

Borja Sotomayor

Table of Contents

Introduction

Welcome to the Globus Toolkit 3 Programmer's Tutorial! This document is intended as a starting point for anyone who is going to program grid-based applications using the Globus Toolkit 3 (GT3). We also hope experienced GT3 programmers will find it useful to learn about the more advanced aspects of GT3 and Grid Services.

The tutorial is divided into 3 main areas:

- **Getting Started**: An introduction to key concepts related with Grid Services and GT3.
- **GT3 Core**: A guide to programming basic Grid Services which only use the core services in GT3.
- **GT3 Security Services**: A guide to programming *secure* Grid Services which use the toolkit's Security Services.

Future versions of the tutorial will include sections related to GT3 Higher-Level Services (programming Grid Services which use GT3 services such as Index Service, Job Management, File Transfer, etc.)

GT3 Prerequisite Documents

This tutorial has no GT3 prerequisite documents, since it is intended as a starting point for GT3 programmers. However, you should already be familiar with Grid Computing. The following book can help you get up to speed: The Grid: Blueprint for a New Computing Infrastructure [http://www.amazon.com/exec/obidos/ASIN/1558604758/o/qid=958665349/sr=2-1/103-6896860-5655839] (Edited by Ian Foster and Carl Kesselman). 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 be even more interested in the second edition released in 2003, including tons of new material: The Grid 2: Blueprint for a New Computing Infrastructure [http://www.mkp.com/grid2] (Edited by Ian Foster and Carl Kesselman)

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 GT3.

- 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 wish to program grid-based applications with GT3. Readers who have absolutely no experience with Web Services or the Globus Toolkit should read the whole document. Readers who have some experience with GT3 can safely skip most of the introductory material.

Assumptions

The following knowledge is assumed:

- Programming in Java. If you don't know Java, you can find some useful links here. 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 here.
- 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 program Grid Services.
- Globus Toolkit 2

Related Documents

The Globus Toolkit includes some very useful documents. The ones most related to this document are:

- Java User's Guide: `$GLOBUS_LOCATION/docs/users_guide.html`
- Java Programmer's Guide: `$GLOBUS_LOCATION/docs/java_programmers_guide.html`
- Programmer's API: `$GLOBUS_LOCATION/docs/api/index.html`

Substitute `$GLOBUS_LOCATION` for the root of your GT3 installation. A team at IBM lead by Luis Ferreira has written a thorough Redpaper titled GT3 Quick Start [http://publib-b.boulder.ibm.com/Redbooks.nsf/RedpaperAbstracts/redp3697.html?Open] which explains the GT3 installation process in detail.

GT3 users have also contributed installation and programming guides:

- From Zero to GT3 [http://www-pnp.physics.ox.ac.uk/~stokes/twiki/bin/view/DIRAC/GT3Express]. Written by Ian Stokes-Rees.
- Grid Install for Windows 2000 Platform [http://www.bigdogsoftware.org/]. Written by Michael Schneider.

Once you've become a Grid Services expert, you might have to occasionally take a look at the OGSI specification, available at the OGSI Working Group [http://forge.gridforum.org/projects/ogsi-wg] page.

Document Conventions

The following conventions will be observed in this document.

Code

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

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 two types of notes in the text: General notes, and warnings.

Note

This is a general note.

This kind of notes are usually used after a block of code to point out 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 GT3 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

Last, but certainly not least, a lot of readers have helped to improve the tutorial by reporting bugs and typos, as well as making very constructive comments and suggestions:

> 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 [http://www.bigdogsoftware.org], Shiva Shankar Chetan, Nelson Sproul, Ian Stokes-Rees, Jason Young, Matthew Vranicar, James Werner

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

Part I. Getting Started

Table of Contents

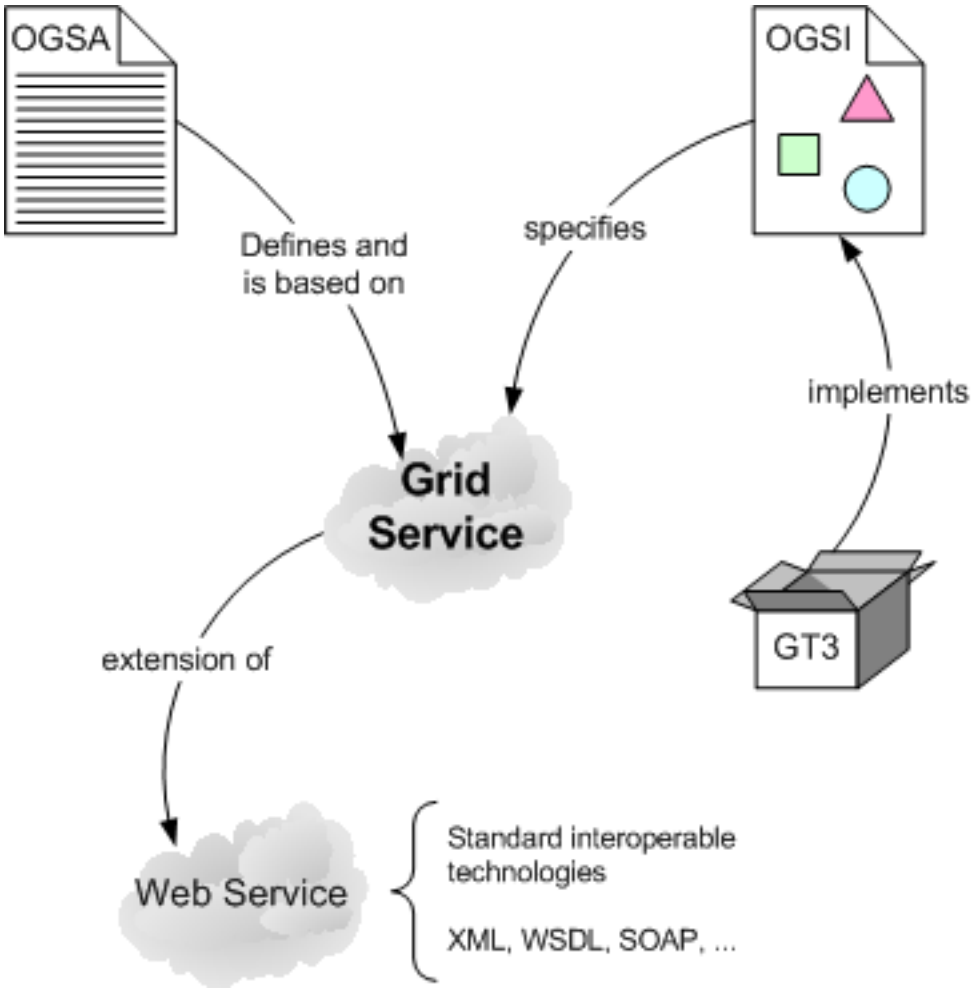
Chapter 1. Key Concepts

There are certain key concepts that must be well understood before being able to program with GT3. This chapter gives a brief overview of all those fundamental concepts.

- **OGSA, OGSI, and GT3** : We'll take a look at what these oft-mentioned acronyms mean, and how they are related.
- **Web Services** : OGSA, OGSI, and GT3 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 GT3, 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.
- **Grid Services** : Grid Services are the core of GT3. We take a look at what a Grid Service is, and how it is related to Web Services.
- **The GT3 Architecture** : After seeing both Grid Services and Web Services, we take a look at the whole GT3 architecture, and how Grid Services fit in it.
- **Java & XML** : Finally, if you want to use GT3, 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, OGSI, and GT3

The third and latest version of the Globus Toolkit is based on something called *Grid Services*. Before defining Grid Services, we're going to see how Grid Services are related to a lot of acronyms you've probably heard (OGSA, OGSI, ...), but aren't quite sure what they mean exactly. The following diagram summarizes the major players in the Grid Services world:



OGSA

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 finds in a grid application (job management services, resource management services, security services, etc.) by specifying a set of standard interfaces for these 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 JobSubmisionInterface 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 had several shortcomings which made it inadequate for OGSA's needs. OGSA overcame this obstacle by defining an extended type of Web Service called *Grid Service* (as shown in the diagram: **Grid Services are defined by OGSA**). A Grid Service is simply a Web Service with a lot of extensions that make it adequate for a grid-based application (and, in particular, for OGSA). In the diagram: **Grid Services are an extension**

of Web Services. Finally, since Grid Services are going to be the distributed technology underlying OGSA, it is also correct to say that *OGSA is based on Grid Services.*

OGSI

OGSA alone doesn't go into much detail when describing Grid Services. It basically outlines what a Grid Service should have (that Web Services don't) but little else. That is why OGSA spawned another standard called the Open Grid Services Infrastructure (OGSI, also developed by The Global Grid Forum [http://www.ggf.org]) which gives a formal and technical specification of what a Grid Service is. In other words, for a high-level architectural view of what Grid Services are, and how they fit into the next generation of grid applications, OGSA is the place to go. For an excruciatingly detailed specification of how Grid Services work, OGSI is the place to go. In the diagram: **Grid Services are specified by OGSI** (as opposed to simply 'being defined' by OGSA). The Related Documents section has a link to the Grid Service Specification.

The Globus Toolkit 3

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 third version of the toolkit (GT3) includes a complete implementation of OGSI (in the diagram **GT3 implements OGSI**). However, it's very important to understand that GT3 isn't *only* an OGSI implementation. It includes a whole lot of other services, programs, utilities, etc. Some of them are built *on top of OGSI* and are called the *WS (Web Services) components*, while other are not built on top of OGSI and are called the *pre-WS components*. We'll take a closer look at the GT3 architecture shortly.

I still don't get it: What is the difference between OGSA, OGSI, and GT3?

Consider the following simple example. Suppose you want to build a new house. The first thing you need to do is to hire an architect to draw up the blueprints, so you can get an idea of what your house will look like. Once you're happy with the architect's job, it's time to hire an engineer who will plan all the construction details (where to put the master beams, the power cables, the plumbing, etc.). The engineer then passes all his plans to qualified professional workers (construction workers, electricians, plumbers, etc) who will actually build the house.

We could say that OGSA (the definition) is the blueprints the architect creates to show what the building looks like, OGSI (the specification) is the structural design that the engineer creates to support the architect's vision for the building, and GT3 is the bricks, cement, and beams used to build the building to the engineer's specifications.

A short introduction to Web Services

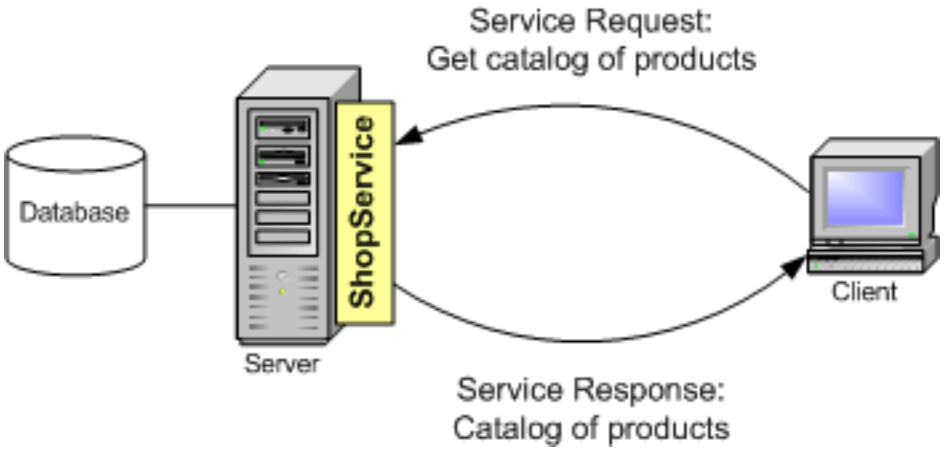
Since Web Services are the basis for Grid Services, understanding the Web Services architecture is fundamental to using GT3 and programming Grid Services.

Lately, 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 have to develop an application for a chain of stores. These stores are all around the country, but my master catalog of products is only available in a database at my central offices, yet the software at the stores must be able to access that catalog. I could *publish* the catalog through a Web Service called *ShopService*.

> IMPORTANT: 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* (the PCs at the store) would then contact the *Web Service* (in the *server*), and send a *service request* asking for the catalog. The server would return the catalog through a *service response*. Of course, this is a very sketchy example of how a Web Service works. In a moment we'll see all the details.



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)

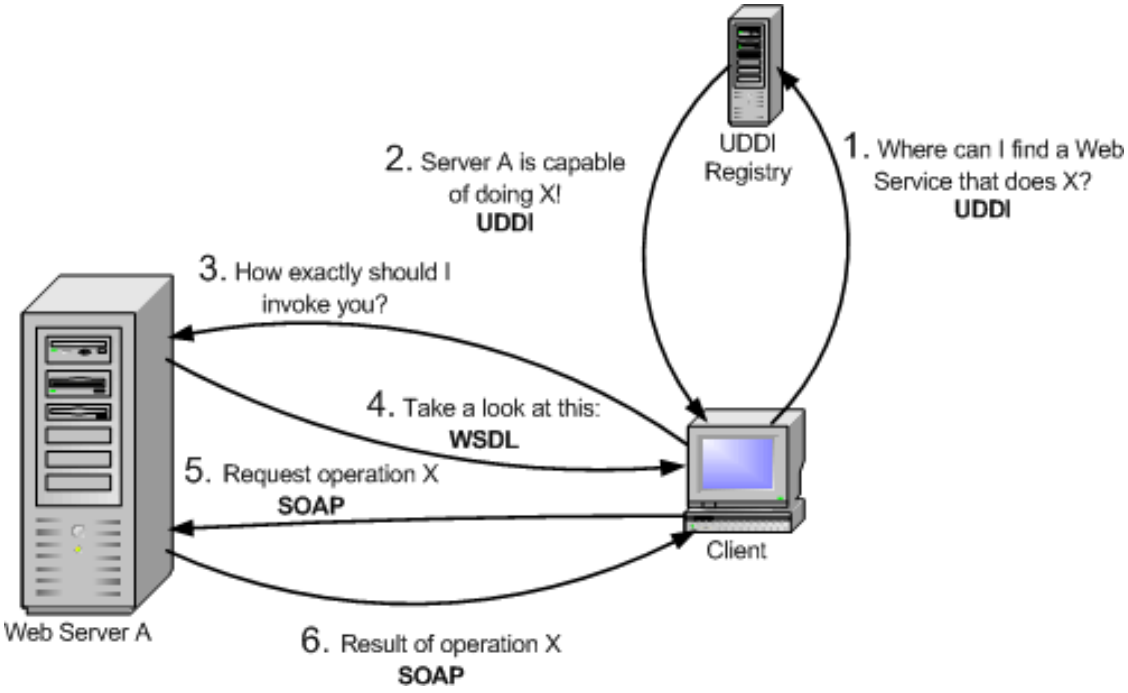
Of course, Web Services also have some disadvantages:

- 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.) In fact, in the next page we'll see that Grid Services actually make up for this lack of versatility.

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.



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 *find* a Web Service that meets our requirements. For example, we might be interested in locating a public Web Service which can give me the temperature in US cities. We'll do this by contacting a UDDI registry.

2. The UDDI registry will reply, telling us what servers can provide us the service we require (e.g. the temperature in US cities)

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 temperature of a US city, buy what is the actual service invocation? The method I have to invoke might be called Temperature getCityTemperature(int CityPostalCode), but it could also be called int getUSCityTemp(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 temperature of a certain city.

6. The Web Service will kindly reply with a *SOAP response* which includes the temperature we asked for, or maybe an error message if our SOAP request was incorrect.

Web Services Addressing

We have just seen a simple Web Service invocation. At one point, the UDDI registry '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.

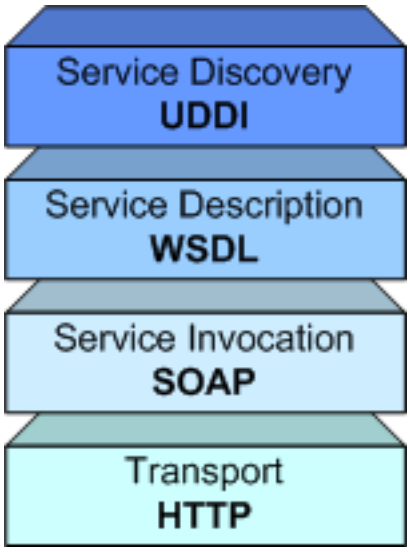
For example, the UDDI registry might have replied with the following URI:

```
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 receive the Grid Service URI as a command-line argument.

Web Services Architecture

Now that we've seen the different players in a Web Service invocation, let's take a closer look at the Web Services Architecture:



- **Service Discovery:** This part of the architecture allows us to find Web Services which meet certain requirements. This part is usually handled by UDDI (Universal Description, Discovery, and Integra-

tion). GT3 currently doesn't include support for UDDI.

- **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.

What a Web Service Application Looks Like

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 programmed with CORBA or RMI, this structure will look pretty familiar.

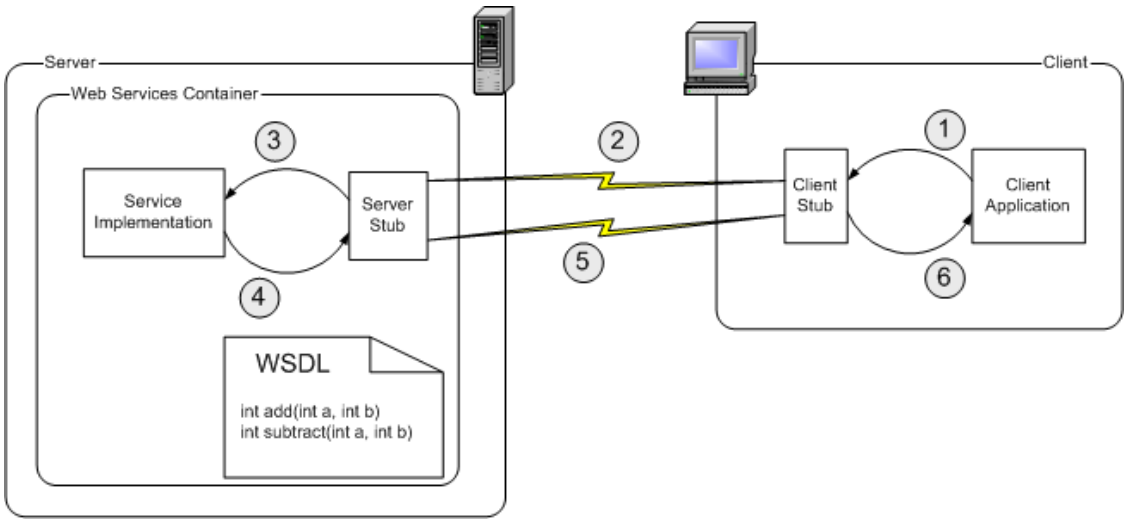
First of all, you should know that despite having a lot of protocols and languages floating around, Web Services programmers usually never write a single line of SOAP or WSDL. Once we've reached a point where our client application needs to invoke a Web Service, we *delegate* that task on a piece of software called a *client stub*. The good news is that there are plenty of tools available that will generate client stubs automatically for us, usually based on the WSDL description of the Web Service.

Therefore, you shouldn't interpret the "Typical Invocation" diagram literally. A Web Services client doesn't usually do all those steps in a single invocation. A more correct sequence of events would be the following:

1. We locate a Web Service that meets our requirements through UDDI.
2. We obtain that Web Service's WSDL description.
3. We generate the stubs *once*, and include them in our application.
4. The application uses the stubs each time it needs to invoke the Web Service.

Programming the server side is just as easy. We don't have to write a complex server program which dynamically interprets SOAP requests and generates SOAP responses. We can simply implement all the functionality of our Web Service, and then generate a *server stub* (the term *skeleton* is also common) which will be in charge of interpreting requests and *forwarding* them to the service implementation. When the service implementation obtains a result, it will give it to the server stub, which will generate the appropriate SOAP response. The server stub can also be generated from a WSDL description, or from other interface definition languages (such as IDL). Furthermore, both the service implementation and the server stubs are managed by a piece of software called the *Web Service container*, which will make sure that incoming HTTP requests intended for a Web Service are directed to the server stub.

So, the steps involved in invoking a Web Service are described in the following diagrams.



Let's suppose that we've already located the Web Service, and generated the client stubs from the WSDL description. Furthermore, the server-side programmer will have generated the server stubs.

1. Whenever the client application needs to invoke the Web Service, it will actually 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 Web Services container 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. The service implementation receives the request from the service stub, and carries out the work it has been asked to do. For example, if we are invoking the int add(int a, int b) method, the service implementation will perform an addition.

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.

By the way, 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.

What is a Grid Service?

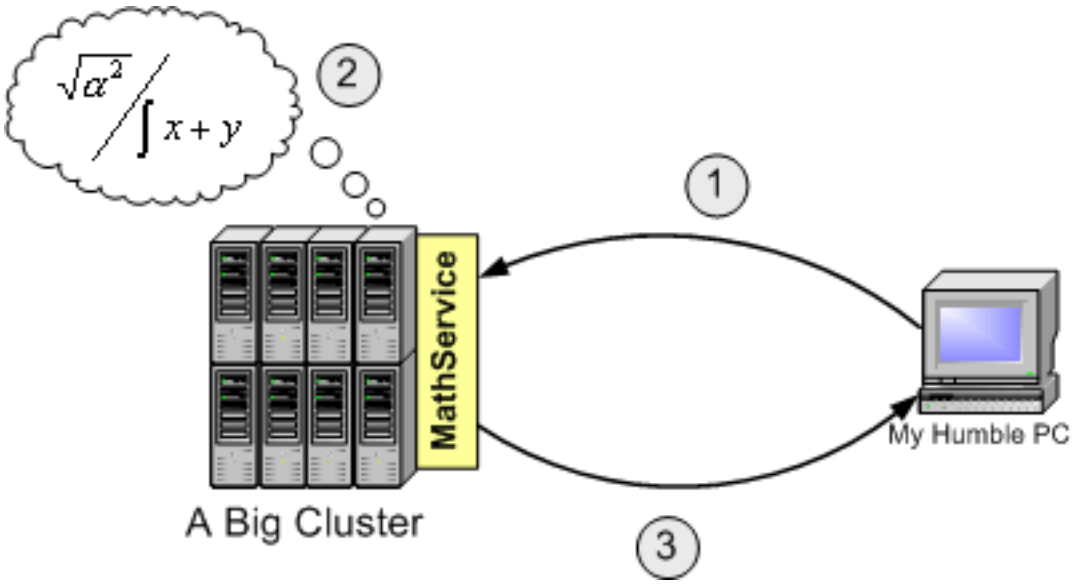
As mentioned before, Web Services are the technology of choice for Internet-based applications with loosely coupled clients and servers. That makes them the natural choice for building the next generation of grid-based applications. However, remember Web Services do have certain limitations. In fact, plain Web Services (as currently specified by the W3C) wouldn't be very helpful for building a grid application. Enter **Grid Services**, which are basically Web Services with improved characteristics and services.

We'll take a brief look at the main improvements introduced in OGSI:

- Stateful and potentially transient services
- Service Data
- Notifications
- Service Groups
- portType extension
- Lifecycle management
- GSH &amp GSR

Stateful and potentially transient services

This first feature is probably one of the most important improvement with regard to Web Services. Let's see what this feature is all about by using a simple example. Imagine your organization has a really big cluster capable of performing the most mind-boggling calculations. However, this cluster is located in your central headquarters in Chicago, and you need employees from your offices in New York, Los Angeles, and Seattle to conveniently use the cluster's computational power. This looks like a perfect scenario for a Web Service!



We could implement a Math Web Service called *MathService* which offered operations such as `SolveReallyBigSystem()`, `SolveFermatsLastTheorem()`, etc. At first, we would be able to perform typical Web Service invocations:

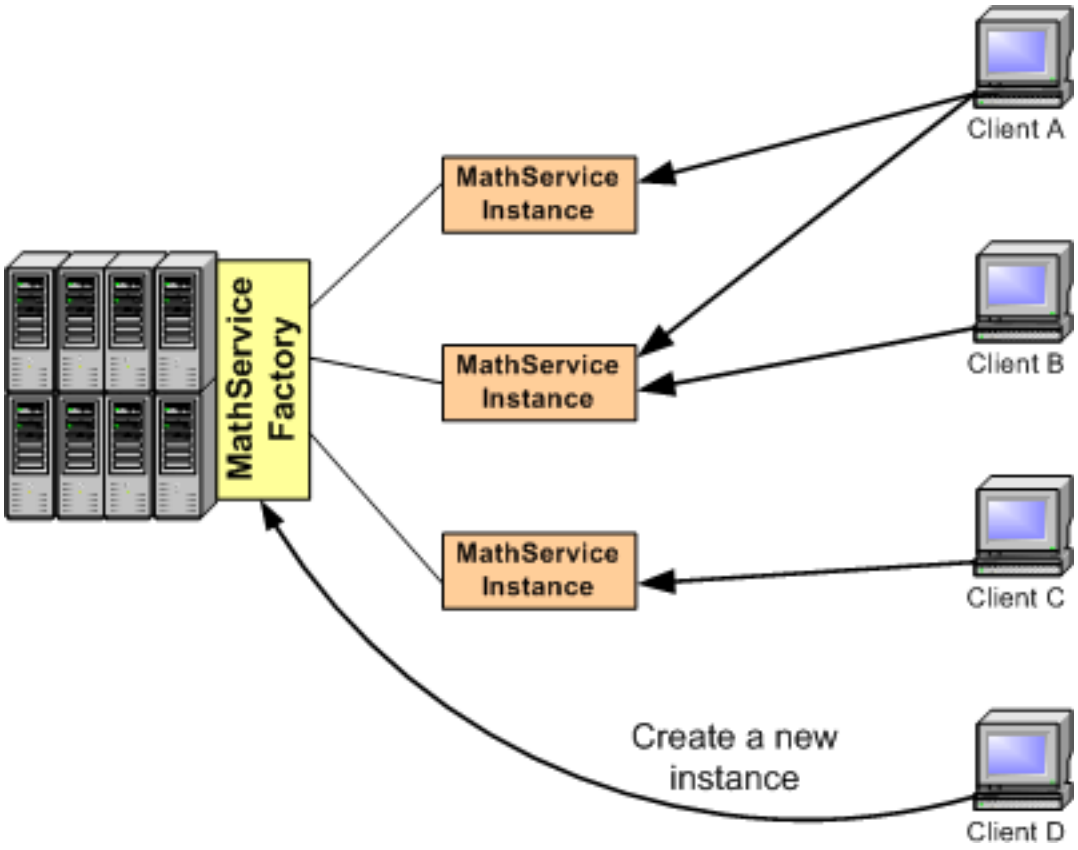
1. Invoke MathService, asking it to perform a certain operation.
2. MathService will instruct the cluster to perform that operation.

3. MathService will return the result of the operation.

So far, so good. However, let's be a bit more realistic. If you're going to access a remote cluster to perform complex mathematical operations, you probably won't perform a single operation, but rather a chain of operations, which will all be related to each other. However, Web Services are *stateless*. "Stateless" means that Web Services can't remember what you've done from one invocation to another. If we wanted to perform a chain of operations, we would have to get the result of one operation and send it as a parameter to the next operation.

Furthermore, even if we solved the stateless problem (some Web Services containers actually work around this problem), Web Services are still *non-transient*, which means that they *outlive* all their clients. Web Services are also referred to as *persistent* (as opposed to *transient*; this doesn't mean 'persistent' in the sense of 'persisting data to secondary storage, a hard drive, etc.') because their lifetime is bound to the Web Services container (a Web Service is available from the moment the server is started, and doesn't go down until the server is stopped) In any case, this implies that, after one client is done using a Web Service, all the information the Web Service is remembering could be accessed by the next clients. In fact, while one client is using the Web Service, another client could access the Web Service and potentially mess up the first client's operations. This certainly isn't a very elegant solution!

Grid Services solve both problems by allowing programmers to use a *factory/instance* approach to Web Services. Instead of having one big stateless MathService shared by all users, we could have a central MathService factory in charge of maintaining a bunch of MathService instances. When a client wants to invoke a MathService operation, it will talk to the instance, not to the factory. When a client needs a new instance to be created (or destroyed) it will talk to the factory.



This diagram shows how there doesn't necessarily have to be one instance per client. One instance could

be shared by two clients, and one client could have access to two instances. These instances are *transient*, because they have a limited lifetime which isn't bound to the lifetime of the Grid Services' container. In other words, we can create and destroy instances at will whenever we need them (instead of having one persistent service permanently available). The actual lifecycle of an instance can vary from application to application. Usually, we'll want instances to live only as long as a client has any use for them. This way, every client has its own personal instance to work with. However, there are other scenarios where we might want an instance to be shared by several users, and to self-destruct after no clients have accessed it for a certain time.

Finally, notice how Grid Services are *potentially* transient. This means that not *all* Grid Services have to use (by definition) a factory/instance approach. A Grid Service can be persistent, just like a normal Web Service. Choosing between persistent Grid Services or factory/instance Grid Services depends entirely on the requirements of your application.

Lifecycle management

Since we are now dealing with services that have non-trivial lifecycles (if we use a factory/instance model, instances can be created and destroyed at any time), lifecycle management mechanisms are provided in Grid Services. OGSI itself only supplies some very basic mechanisms, which are complemented by additional mechanisms in GT3, as we'll see later on.

Service Data

Service Data, along with statefulness and transience, ranks very high in the list of 'the best things Grid Services add to Web Services'. In fact, Service Data is my personal favorite extension!

Service Data allows us to easily include a set of *structured data* to any service, which can then be accessed directly through its interface. Since plain Web Services only allow operations to be included in the WSDL interface, you could think of Service Data as an extension that allows us to include not only operations in the WSDL interface, but also attributes. However, Service Data is much more than simple attributes, since we can easily include any type of data (fundamental types, classes, arrays, etc.)

In general, the Service Data we include in a service will fall into one of two categories:

- **State information**: Provides information on the current state of the service, such as operation results, intermediate results, runtime information, etc.
- **Service metadata**: Information on the service itself, such as system data, supported interfaces, cost of using the service, etc.

If you're not too sure what service data is, don't worry: a much more detailed explanation will be given when we start coding Grid Services with service data.

Notifications

A Grid Service can be configured to be a *notification source*, and certain clients to be *notification sinks* (or subscribers). This means that if a change occurs in the Grid Service, that change is *notified* to all the subscribers (not *all* changes are notified, only the ones the Grid Services programmer wants to). In the MathService example, suppose that all the clients perform certain calculations using a variable called *InterestingCoefficient* which is stored in the Grid Service. Any of the clients can modify that value to improve the overall calculation. However, all clients must be notified of that change when it occurs. We can achieve this easily with the Grid Services notifications.

Later on, we'll see that notifications in OGSI are very closely related to service data.

Service Groups

Any service can be configured to act as a *service group* which aggregates other services. We can easily perform 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'. Although the service group functionality included in OGSI is pretty simple, it is nonetheless the base of more powerful directory services (such as GT3's IndexService) which allow us to group different services together and access them through a single point of entry (the service group).

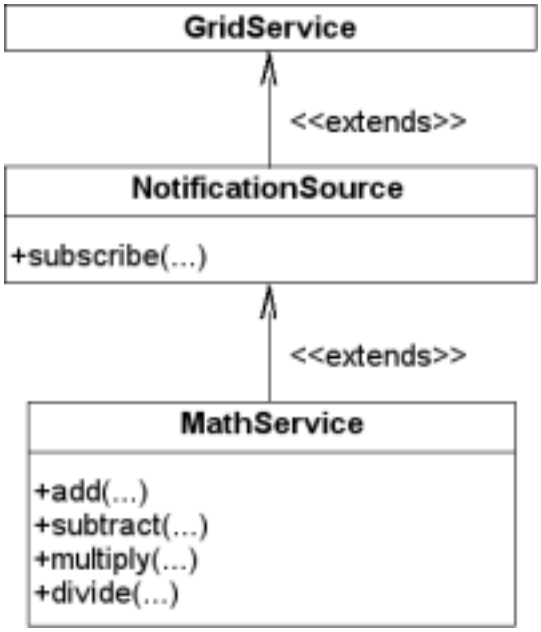
portType extension

In the previous page we saw that a Web Service exposes its interface (the operations it can perform) through a WSDL document. The interface is usually called *portType* (due to a WSDL tag of the same name). A normal Web Service can have only one portType. Grid Services, on the other hand, support *portType extension*, which means we can define a portType as an extension of a previously existing portType.

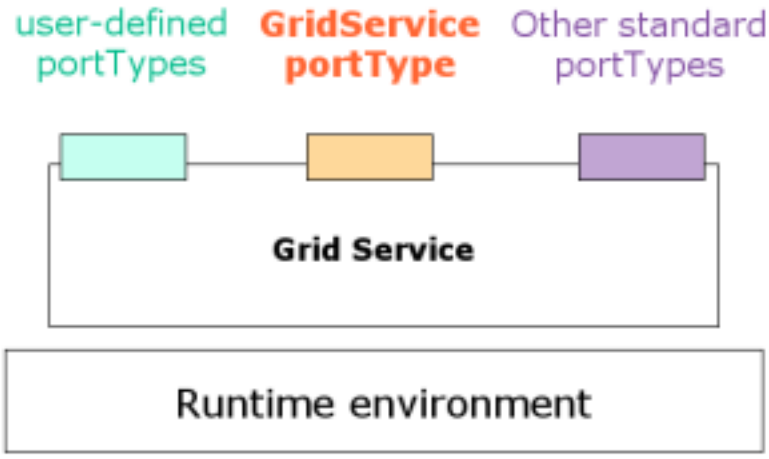
For example, the OGSI specification mandates that all Grid Services *must* extend from a standard portType called GridService:

Thanks to portType extension, we can simply define our own portType as an extension of GridService. With plain web services, we would need to include the declaration of all the operations (including the GridService operations) in a single portType.

Besides the standard GridService portType, OGSI defines a lot of other standard portTypes we can extend from to add functionality to our grid service. For example, there is a NotificationSource portType which we can extend from if we want our service to act as a notification source (we'll take a close look at this portType and other standard portTypes when we start working with code). Notice how NotificationSource itself extends from GridService:



In general, we'll find that grid services can have three types of portTypes:



GSH & GSR

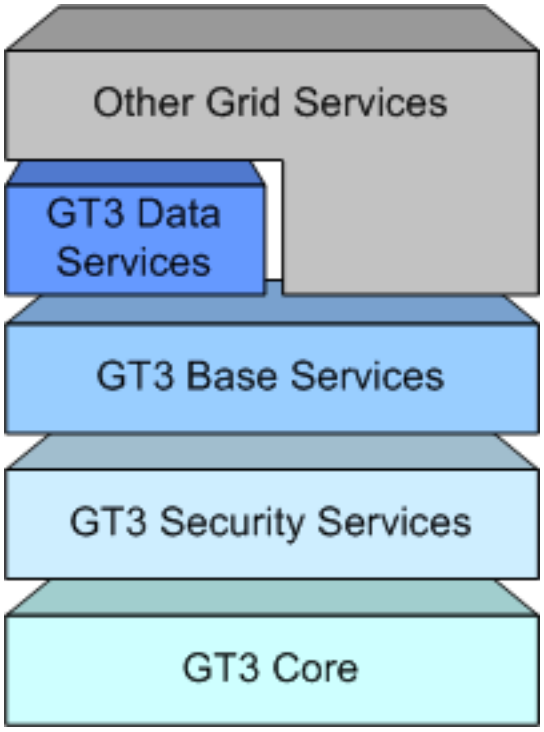
In the previous page we saw that Web Services are addressed with URIs. Since Grid Service *are* Web Services, they are also addressed with URIs. However, OGSI introduces a more powerful addressing scheme.

A "Grid Service URI" is called the *Grid Service Handle*, or simply GSH. Each GSH must be unique. There cannot be two Grid Services with the same GSH. The only problem with the GSH is that it tells me *where* the Grid Service is, but doesn't give me any information on *how* to communicate with the Grid Service (what methods it has, what kind of messages it accepts/receives, etc.). To do this, we need the *Grid Service Reference*, or GSR. In theory, the GSR can take many different forms, but since we will usually use SOAP to communicate with a Grid Service, the GSR will be a WSDL file (remember that WSDL *describes* a Web Service: what methods it has, etc.). In fact, in this tutorial we will only handle

WSDL as a GSR format.

The Globus Toolkit 3

Grid Services sound great, don't they? 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. Grid Services are only a small (but important!) part of the whole GT3 Architecture, which offers developers plenty of services to get serious with Grid programming.



OGSI (i.e. "Grid Services") is the 'GT3 Core' layer. Let's take a look at the rest of the layers from the bottom up:

- **GT3 Security Services**: Security is an important factor in grid-based applications. GT3 Security Services can help us restrict access to our Grid Services, so only authorized clients can use them. For example, we said that only our New York, Los Angeles, and Seattle offices could access MathService. We want to make sure only those offices have access to MathService. Besides the usual security measures (putting the web server behind a firewall, etc.) GT3 gives us one more layer of security with technologies such as SSL and X.509 digital certificates.

- **GT3 Base Services**: This layer actually includes a whole lot of interesting services:

  - **Managed Job Service**: Suppose some particular operation in MathService might take hours or even days to be done. Of course, we don't want to simply stand in front of a computer waiting for the result to arrive (specially if, after 8 hours of waiting, all we get might simply be an error message!) We need to be able to check on the progress of the operation periodically, and have some control over it (pause it, stop it, etc.) This is usually called *job management* (in this case, the term 'job' is used instead of 'operation').

  - **Index Service**: Remember from A short introduction to Web Services that we usually know

what type of Web Service we need, but we have no idea of where they are. This also happens with Grid Services: we might know we need a Grid Service which meets certain requirements, but we have no idea of what its location is. While this was solved in Web Services with UDDI, GT3 has its own Index Service. For example, we could have several dozen MathServices all around the country, each with different characteristics (some might be better suited for statistical analysis, while others might me better for performing simulations). Index Service will allow us to query what MathService meets our particular requirements.

    - **Reliable File Transfer (RFT) Service**: This service allows us to perform large file transfers between the client and the Grid Service. For example, suppose we have an operation in MathService which has to crunch several gigabytes of raw data (for a statistical analysis, for example). Of course, we're not going to send all that information as parameters. We'll be able to send it as a file. Furthermore, RFT guarantees the transfer will be reliable (hence its name). For example, if a file transfer is interrupted (due to a network failure, for example), RFT allows us to restart the file transfer from the moment it broke down, instead of starting all over again.

- **GT3 Data Services**: This layer includes *Replica Management*, which is very useful in applications that have to deal with very big sets of data. When working with large amount of data, we're usually not interested in downloading the whole thing, we just want to work with a small part of all that data. Replica Management keeps track of those *subsets* of data we will be working with.

- **Other Grid Services**: Other non-GT3 services can run on top of the GT3 Architecture.

WSRF & GT4

Before moving on to the practical part of the tutorial, I think it's definitely worthwhile to take five minutes to reading about the next step in the evolution of grid standards: WSRF and GT4.

After having read all the theory behind grid services, you might be thinking: "Gee, Grid Services are really cool!" You might even be thinking: "Why would anyone want to continue using Web Services when you can use hip and cool Grid Services instead?" Well, it so turns out that OGSI was created with the hopes that it would eventually converge with Web Services standards and that, in fact, Web Services and Grid Services would become the same thing. However, however 'hip and cool' OGSI Grid Services might seem, they do in fact have several drawbacks which have blocked this convergence.

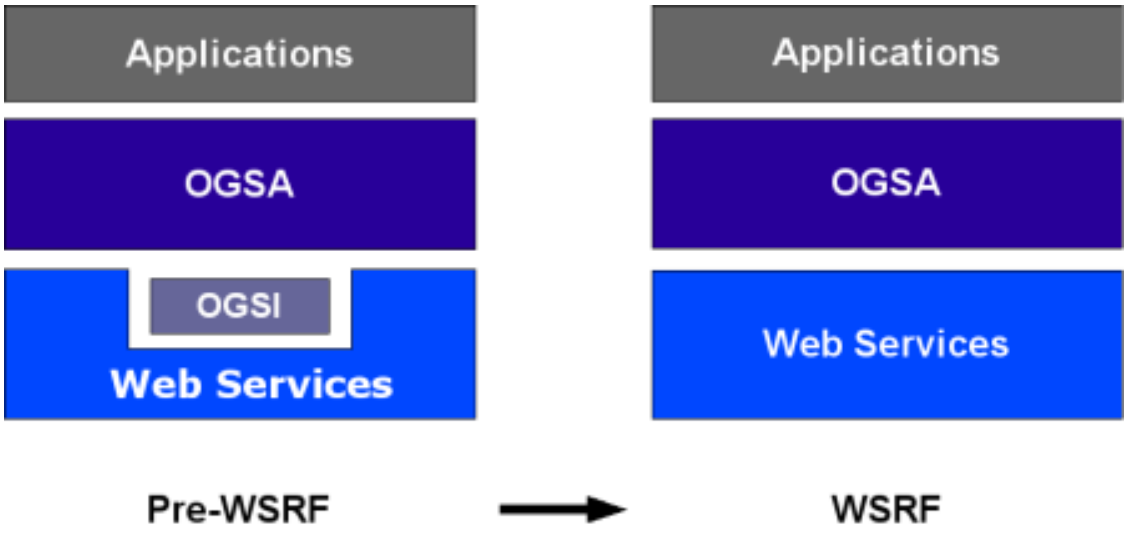
- **The OGSI specification is long and dense**. Yes, practically all specifications are long and dense but, believe me, OGSI is *specially* long and dense.

- **OGSI does not work well with current Web Services tooling.**

- **Too object oriented**. Despite the fact that many Web Services systems have object oriented implementations, web services themselves are not supposed to be object oriented. OGSI, on the other hand, takes a lot of concepts from OO (such as statefulness, the factory/instance model, etc.)

To solve OGSI's problems once and for all, and to improve Grid Services' chances of finally converging with Web Services, a new standard was presented in January 2004 to substitute OGSI: The Web Services Resource Framework [http://www.globus.org/wsrf], or WSRF.

As the following diagram shows, WSRF aims to integrate itself into the family of Web Services standards, instead of simply being a 'patch over Web Services' like OGSI was. In this new scenario, OGSA will be based *directly* on Web Services instead of being based on OGSI Grid Services.



Notice how this new standard doesn't really affect OGSA all that much. All the high level services defined in OGSA (job management, resource management, security services, etc.) will keep having the same interfaces and specifications. The only difference is that the underlying middleware will be pure Web Services instead of OGSI Grid Services. From the standpoint of *defining* the standard interfaces and behavior of (for example) a job submission service, this isn't really such a big change. It will, of course, affect those who want to *implement* (or have already begun implemented) OGSA services.

And how exactly does WSRF intend to gain true convergence with Web Services? Well, first of all, it overcomes OGSI's drawbacks:

- **The OGSI specification is long and dense**. The WSRF specification is divided into 5 documents, plus one complementary specification which is not strictly a part of WSRF (WS-Notification)
- **OGSI does not work well with current Web Services tooling**. WSRF takes into account most of the objections posed by the Web Services community, making it easier for current Web Services tooling to evolve towards including WSRF support.
- **Too object oriented**. WSRF cleanly separates the service from the state, since pure Web Services cannot have state. Support for factory/instance services also disappears, although it can still be implemented by using a design pattern.

The Globus Toolkit 4

Yes, despite the fact that what you're reading is the Globus Toolkit *Three* Programmer's Tutorial, there are already plans for a *fourth* version. This new version, in fact, will be the first available WSRF implementation. The first stable release is expected in the third quarter of 2004.

Don't Panic

At this point, alarm bells might be ringing loudly in your head. If the next version of the Globus Toolkit is just around the corner (with a new standard), then what is the point of spending a single nanosecond on GT3 and OGSI? Well, here's the good news: WSRF and OGSI are *conceptually* the same thing. The main differences between OGSI and WSRF are not 'big' differences (such as a paradigm shift). They are mainly syntactical in nature: WSRF is a refactoring of OGSI, taking into account all the comments from

both the Grid community and the Web Services community, resulting in a more polished and stable standard. In fact, there's even a document available in the WSRF website that shows how there's a direct mapping from OGSI features to WSRF features.

So, what's the point in learning OGSI and GT3? Because the switch from OGSI to WSRF will be very simple if you've already gotten the hang of all the fundamental concepts: service data, notifications, life-cycle management, etc. If you wait for GT4 to come out, you'll have to learn all the theory then. It's like learning to program: Once you've mastered one programming language, moving to a new language is simply a question of learning the syntax of the new language (along with any particular idiosyncrasies that language might have)

Ok, so that previous paragraph might sound like a shameless pitch to get you to read the tutorial :-) Believe me, it's not. If you learn GT3 now instead of waiting for GT4, you'll be able to laugh at those who didn't learn GT3. Like this: "Ha, ha, ha! I scoff thine minuscule knowledge of service data and notifications! Tremble upon my encyclopedic knowledge of service-oriented grid applications!"

Where to learn Java & XML

After seeing all the theory behind GT3, 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

This tutorial currently doesn't include an installation guide. I hope to include one in the future but, in the meanwhile, there are plenty of good GT3 installation guides available on the Web:

- Java User's Guide. The official installation guide from Globus. See Related Documents.
- GT3 Quick Start [http://publib-b.boulder.ibm.com/Redbooks.nsf/RedpaperAbstracts/redp3697.html?Open]. Redpaper written by an IBM team lead by Luis Ferreira.
- From Zero to GT3 [http://www-pnp.physics.ox.ac.uk/~stokes/twiki/bin/view/DIRAC/GT3Express]. Written by Ian Stokes-Rees
- Grid Install for Windows 2000 Platform [http://www.bigdogsoftware.org/]. Written by Michael Schneider.

Part II. GT3 Core

Table of Contents

Chapter 3. Writing Your First Grid Service in 5 Simple Steps

MathService

In this chapter we are going to write and deploy a simple Grid Service. Our first Grid Service is an extremely simple *Math Grid Service*, which we'll refer to as *MathService*. It will allow users to perform the following operations:

- Addition
- Subtraction

High-tech stuff, huh? Don't worry if this seems a bit lackluster. Since this is going to be our first Grid Service, it's better to start with a small didactic service which we'll gradually improve by adding service data, notifications, etc. You should always bear in mind that MathService is, after all, just a means to get acquainted with GT3. Typical grid services are generally much more complex and do more than expose trivial operations (such as addition and subtraction). Although the tutorial is currently based only on MathService, future versions of the tutorial will include examples of grid services which you could find in 'real' applications.

The Five Steps

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

Don't worry if you don't understand these five steps or are baffled by terms such as GWSDL, 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. However, the rest of the examples will simply instruct you to perform a step, and 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 Grid 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://www.casa-sotomayor.net/gt3-tutorial]. The tutorial 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 filesystem and untar-ungzip the file there. From now on, we'll refer to that directory as **$TUTORIAL_DIR**.

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

- **With the tutorial source files**: You'll have all the source code (Java, GWSDL, and WSDD) ready to use in `$TUTORIAL_DIR`, so there's no need to manually modify these files.
- **Without the tutorial source files**: Some people don't like getting all the source code ready to use out-of-the-box, but rather prefer to write the files themselves so they can have a better understanding of what they're doing at each point. In fact, I think this is probably the best way to follow this first part of the tutorial. Since this first part includes complete code listing in the tutorial (which you can copy and paste to a file), you can easily write all the files yourself. However, you *do* need a set of auxiliary files included in the tutorial bundle which are needed to build and deploy the services. So, if you want to follow the tutorial without the source files, you still need to download the tutorial files. Once you're in `$TUTORIAL_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 GWSDL

The first step in writing a Grid Service (or a Web Service) 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, what databases it will access, etc.) We just need to know what *operations* will be available to our users. Remember that, in Web/Grid Services lingo, the service interface is usually called the *port type* (usually written *portType*).

As we saw in A short introduction to Web Services, 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. However, this is not entirely true. In GT3 we have two options:

- **Writing the WSDL directly**. This is the most versatile option. If we write WSDL directly, we have total control over the description of our service's portType. However, it is not the most user-friendly one, since WSDL is a rather verbose language.
- **Generating WSDL from a Java interface**. We can generate WSDL automatically from a Java interface. This is the easiest option, but not the most versatile (since very complicated interfaces are not always converted correctly to WSDL).

At first sight, it might seem that starting with an interface language (such as Java interface or an IDL interface) might be the best option, since it is the most user-friendly. 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 getValue();
```

```
}
```

...and we'd be finished with step 1! 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.

Actually, what we're going to write is not WSDL, but Grid WSDL (or GWSDL), which is an extension of WSDL used in the OGSI specification (and, therefore, in the Globus Toolkit). This 'extended' WSDL supports all the features described in the What is a Grid Service? page which are not supported in plain Web Services (and, therefore, in plain WSDL). In a moment we'll see the exact differences between WSDL and GWSDL. Before that, we'll take a good look at the GWSDL 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 GWSDL file, but rather to present the GWSDL file for this particular example and explain the main differences between WSDL and GWSDL. If you have no idea whatsoever of how to write WSDL, now is a good time to take a look at the following section of the How to... appendix: How to write a GWSDL description of your Grid Service.

A general description of the interface

The GWSDL code we'll be using is the equivalent of the Java interface shown above. However, that interface might raise a couple of eyebrows. If we said that our MathService is going to allow addition and subtraction, why do `add` and `subtract` receive only *one* parameter? Did the author of the tutorial flunk lower school math? Addition and subtraction involves two numbers! And what's with that `getValue` method? An explanation is in order...

The reason why addition and subtraction have one parameter is because we're going to use our service as an *accumulator* to demonstrate how grid services are *stateful* (remember: stateful means the service remembers internal values from one call to the other). MathService will have an internal value which will initially be zero. Each call to `add` and `subtract` will modify that internal value, which we'll be able to access thanks to the `getValue` method. This will allow us to observe that the service remembers that internal value from once call to the next (unlike a plain *stateless* web service, which would be unable to 'remember' that value).

The GWSDL code

Ok, so supposing you either know WSDL or have visited the How to write a GWSDL description of your Grid Service page, take a good thorough look at this GWSDL code:

```
<?xml version="1.0" encoding="UTF-8"?>
<definitions
name="MathService"
  targetNamespace="http://www.globus.org/namespaces/2004/02/progtutorial/MathServi
  xmlns:tns="http://www.globus.org/namespaces/2004/02/progtutorial/MathService"
  xmlns:ogsi="http://www.gridforum.org/namespaces/2003/03/OGSI"
  xmlns:gwsdl="http://www.gridforum.org/namespaces/2003/03/gridWSDLExtensions"
  xmlns:xsd="http://www.w3.org/2001/XMLSchema"
  xmlns="http://schemas.xmlsoap.org/wsdl/">

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

```
<types>
<xsd:schema
targetNamespace="http://www.globus.org/namespaces/2004/02/progtutorial/MathService
  attributeFormDefault="qualified"
  elementFormDefault="qualified"
  xmlns="http://www.w3.org/2001/XMLSchema">
  <xsd:element name="add">
    <xsd:complexType>
      <xsd:sequence>
        <xsd:element name="value" type="xsd:int"/>
      </xsd:sequence>
    </xsd:complexType>
  </xsd:element>
  <xsd:element name="addResponse">
    <xsd:complexType/>
  </xsd:element>
  <xsd:element name="subtract">
    <xsd:complexType>
      <xsd:sequence>
        <xsd:element name="value" type="xsd:int"/>
      </xsd:sequence>
    </xsd:complexType>
  </xsd:element>
  <xsd:element name="subtractResponse">
    <xsd:complexType/>
  </xsd:element>
  <xsd:element name="getValue">
    <xsd:complexType/>
  </xsd:element>
  <xsd:element name="getValueResponse">
    <xsd:complexType>
      <xsd:sequence>
        <xsd:element name="value" type="xsd:int"/>
      </xsd:sequence>
    </xsd:complexType>
  </xsd:element>
</xsd:schema>
</types>

<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="GetValueInputMessage">
  <part name="parameters" element="tns:getValue"/>
</message>
<message name="GetValueOutputMessage">
  <part name="parameters" element="tns:getValueResponse"/>
</message>

<gwsdl:portType name="MathPortType"
extends="ogsi:GridService">
  <operation name="add">
    <input message="tns:AddInputMessage"/>
    <output message="tns:AddOutputMessage"/>
    <fault name="Fault" message="ogsi:FaultMessage"/>
```

```
  </operation>
  <operation name="subtract">
    <input message="tns:SubtractInputMessage"/>
    <output message="tns:SubtractOutputMessage"/>
    <fault name="Fault" message="ogsi:FaultMessage"/>
  </operation>
  <operation name="getValue">
    <input message="tns:GetValueInputMessage"/>
    <output message="tns:GetValueOutputMessage"/>
    <fault name="Fault" message="ogsi:FaultMessage"/>
  </operation>
</gwsdl:portType>

</definitions>
```

Note

This file is `$TUTORIAL_DIR/schema/progtutorial/MathService/Math.gwsdl`. If you're wondering why you have to save it in that particular directory, take a look at the Tutorial Directory Structure appendix.

If you know WSDL, you'll recognize this as a pretty straightforward WSDL file which defines three operations: add, subtract, and getValue (along with all the necessary messages and types). Let's take a closer look at some important parts of the code. First of all, notice how the target namespace for the Grid Service is:

```
http://www.globus.org/namespaces/2004/02/progtutorial/MathService
```

Furthermore, we have to declare the following OGSI namespaces:

```
xmlns:ogsi="http://www.gridforum.org/namespaces/2003/03/OGSI"
xmlns:gwsdl="http://www.gridforum.org/namespaces/2003/03/gridWSDLExtensions"
```

We also have to import a GWSDL file that defines all the OGSI-specific types, messages, and portTypes.

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

Finally, notice how we have no bindings whatsoever in our GWSDL file (bindings are an essential part of a normal WSDL file). We don't have to add them manually, since they are generated automatically with a GT3 tool.

Namespace mappings

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

going to be implemented.

However, there will of course come a moment when I'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 where described in A short introduction to Web Services [../../start/key/web_services.html]) which are generated from the GWSDL file using a GT3 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 GWSDL namespaces to Java packages:

```
http\://www.globus.org/namespaces/2004/02/progtutorial/MathService=
        org.globus.progtutorial.stubs.MathService

http\://www.globus.org/namespaces/2004/02/progtutorial/MathService/bindings=
        org.globus.progtutorial.stubs.MathService.bindings

http\://www.globus.org/namespaces/2004/02/progtutorial/MathService/service=
        org.globus.progtutorial.stubs.MathService.service
```

Note

Each mapping must go in one line (i.e. the above file should have *three* lines). This file is `$TUTORIAL_DIR/namespace2package.mappings`.

The first namespace is the target namespace of the GWSDL file. The other two namespaces are automatically generated when a GT3 tool 'completes' the GWSDL file (including the necessary bindings). For all the tutorial, we'll be placing all the stub classes in the following Java package:

```
org.globus.progtutorial.stubs
```

Since we're defining a service called MathService, we're specifically mapping the GWSDL file to the following package:

```
org.globus.progtutorial.stubs.MathService
```

However, take into account that the stubs classes are *generated* from the GWSDL 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.progtutorial.stubs` package in `$TUTORIAL_DIR`, because you won't find them there. If you are of a curious predisposition, 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.

Differences between WSDL and GWSDL

Remember that this is GWSDL, an extension of WSDL. Actually, this is not entirely true. GWSDL has certain features that WSDL 1.1 doesn't have, but which will be available in WSDL 1.2/2.0. Since WSDL 1.2/2.0 is still a W3C Working Draft (in other words, not a stable standard, and bound to change in the near future), the Global Grid Forum [http://www.ggf.org/] was unable to use WSDL 1.2/2.0 (with all its great Grid-friendly improvements) in OGSI. So, they created GWSDL as a *temporary* solution. In fact, the Global Grid Forum has said that, as soon as WSDL 1.2/2.0 is a W3C Recommendation (a stable standard), it will substitute GWSDL for WSDL 1.2/2.0. However, since the announcement of WSRF, this has changed (as OGSI will be discontinued when WSRF appears). WSRF will not use GWSDL, but it won't wait for WSDL 1.2/2.0 either, so it will use pure WSDL 1.1 for interface description.

So, what exactly are the improvements in GWSDL? Well, they're basically related to the improvements that Grid Services introduce with respect to Web Services (as seen in What is a Grid Service? [../../start/key/grid_service.html]). If you are WSDL-literate, you've probably already spotted one improvement in the above code:

```
<gwsdl:portType name="MathPortType" extends="ogsi:GridService">

</gwsdl:portType>
```

First of all, notice how we're not using the WSDL <portType> tag, but a tag from the GWSDL namespace: <gwsdl:portType>. This new tag has an extends attribute. This is the first major improvement in GWSDL: portType extension. You can define a PortType as an extension of one of more portTypes. In this case, we're extending from an OGSI PortType called GridService (all Grid Services must implement this interface).

The second major improvement is related to Service Data, which we will see in one of the following chapters.

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 implementation of a Grid Service is simply a Java class which, besides implementing the operations described in the GWSDL file, has to meet certain requirements. This class can furthermore include additional *private* methods which won't be available through the Grid Service interface, but which our service can use internally.

We'll implement the service in a Java class called MathImpl which we'll place in `$TUTORIAL_DIR/org/globus/progtutorial/services/core/first/impl/MathImpl.java`. If you're wondering why we're saving it in that particular spot, take a look at the Tutorial Directory Structure appendix. However, if you don't completely grasp the directory structure right now, don't worry. You can safely take a leap of faith right now, save the files where the tutorial tells you to and, after you've done a couple of examples, take a look at the Tutorial Directory Structure and see how everything fits together.

So, let's start coding! The first thing we'll do in the Java file is include the package declaration, and import a couple of classes we'll need (and which will be described shortly).

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

import org.globus.ogsa.impl.ogsi.GridServiceImpl;
import org.globus.progtutorial.stubs.MathService.MathPortType;
import java.rmi.RemoteException;
```

Now we'll declare the MathImpl class, which will be the implementation of our Grid Service.

```
public class MathImpl extends GridServiceImpl implements MathPortType
```

Notice the following:

- MathImpl is a child class of GridServiceImpl. All Grid Services must extend from the base class GridServiceImpl. This is what is usually called the *skeleton* class, because it contains the 'bare bones' -the basic functionality- common to all Grid Services. In fact, GridServiceImpl implements all the operations declared in the GridService portType which *all* Grid Services must extend from (as described in What is a Grid Service?)
- Our Grid Service implements an interface named MathPortType. Remember that *PortType* is another name for "service interface". We're telling Java that this Grid Service implements a particular portType: the Math PortType. But, where did MathPortType come from? This is one of the *stub files* which will be generated from the GWSDL file once we compile the service..

Next, we have to write a constructor for our Grid Service. Our constructor will simply call the GridServiceImpl constructor, which expects a String with a description of the Grid Service.

```
public MathImpl()
{
        super("Simple Math Service");
}
```

Next, remember we said in the previous page that we want MathService to 'remember' an internal value which we'll add to and subtract from using the add and subtract methods. We simply have to declare a private integer attribute:

```
private int value = 0;
```

Finally, we implement the methods specified in the service interface: add, subtract, and getValue. Notice how, despite being very simple methods, they can *all* throw a RemoteException. Since they are remote methods (methods which can be accessed remotely, through the Grid Service), a RemoteException can be thrown if there is a problem between the server and the client (for example, if there is a network error).

```
public void add(int a) throws RemoteException
{
  value = value + a;
}

public void subtract(int a) throws RemoteException
{
  value = value - a;
}

public int getValue() throws RemoteException
{
  return value;
}
```

With that, we have finished implementing our Grid Service. Now we only have a couple of simple steps before we can finally see it work!

The complete code for the implementation is shown here:

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

import org.globus.ogsa.impl.ogsi.GridServiceImpl;
import org.globus.progtutorial.stubs.MathService.MathPortType;
import java.rmi.RemoteException;

public class MathImpl extends GridServiceImpl implements MathPortType
{
  private int value = 0;

  public MathImpl()
  {
    super("Simple MathService");
  }

  public void add(int a) throws RemoteException
  {
    value = value + a;
  }

  public void subtract(int a) throws RemoteException
  {
    value = value - a;
  }

  public int getValue() throws RemoteException
  {
    return value;
  }
}
```

Note

This file is `$TUTORIAL_DIR/org/globus/progtutorial/services/core/first/impl/MathImpl.java`

Step 3: Configuring the deployment in WSDD

Up to this point, we have written the two most important parts of a Grid Service: the service interface (GWSDL) and the service implementation (Java). Now, we somehow have to put all these pieces together, and make them available through a Grid Services-enabled web server! This step is called the *deployment* of the Grid Service.

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

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

  <service name="progtutorial/core/first/MathService" provider="Handler" style="wr
    <parameter name="name" value="MathService"/>
    <parameter name="className" value="org.globus.progtutorial.stubs.MathService.M

    <parameter name="baseClassName" value="org.globus.progtutorial.services.core.f
    <parameter name="schemaPath" value="schema/progtutorial/MathService/Math_servi

    <!-- Start common parameters -->
    <parameter name="allowedMethods" value="*"/>
    <parameter name="persistent" value="true"/>
    <parameter name="handlerClass" value="org.globus.ogsa.handlers.RPCURIProvider"
  </service>

</deployment>
```

Note

This file is `$TUTORIAL_DIR/org/globus/progtutorial/services/core/first/server-deploy.wsdd`. If you're using the provided example files, you'll notice there is an extra `<service>` tag. You can safely ignore it for now. It will be explained later on.

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

The 'service name'

```
<service name="progtutorial/core/first/MathService" provider="Handler" style="wrap
```

This specifies the location where our Grid Service will be found. If we combine this with the base address of our Grid Service container, we will get the full GSH of our Grid Service. For example, if we are using the GT3 standalone container, the base URL will probably be http://localhost:8080/ogsa/services. Therefore, our service's GSH would be:

```
http://localhost:8080/ogsa/services/progtutorial/core/first/MathService
```

The 'service name' (again)

```
<parameter name="name" value="MathService"/>
```

Although this might seem confusing, the <service> element has both a name attribute (the one seen above) and name parameter. The parameter simply contains a brief description of the service (e.g."MathService")

className and baseClassName

```
<parameter name="className" value="org.globus.progtutorial.stubs.MathService.MathP
<parameter name="baseClassName" value="org.globus.progtutorial.services.core.first
```

In strict WSDD, this parameter would refers to the class which implements the service interface (in our case, MathImpl from the previous page). However, notice how the value of this parameter is the MathPortType stub interface mentioned in the previous page.

Since Grid Services are a tad more complex that plain web services, it's necessary to distinguish between the className and the baseClassName. The className refers to the interface that exposes all the functionality of the grid service (MathPortType), and baseClassName is the class that provides the implementation for our grid service. In our case, baseClassName is the MathImpl class we implemented in the previous page. When we see operation providers (in the next section) we'll see that the baseClassName doesn't need to include an implementation for *all* the methods exposed in the className (we'll see that it's possible to delegate some or all of the our methods to special classes called *operation providers*)

The WSDL file

```
<parameter name="schemaPath" value="schema/progtutorial/MathService/Math_service.w
```

The schemaPath tells the grid services container where the WSDL file for this service can be found. No, that's not a typo, I said *WSDL*, not *GWSDL*. Remember that GWSDL is a (non-standard) extension of WSDL, so it must first be converted to WSDL so it can be truly interoperable with existing web services technologies. This WSDL file (`Math_service.wsdl`) will be generated automatically by a GT3 tool when we compile the service.

The common parameters

```
<!-- Start common parameters -->
<parameter name="allowedMethods" value="*"/>
<parameter name="persistent" value="true"/>
<parameter name="handlerClass" value="org.globus.ogsa.handlers.RPCURIProvider"/>
```

These are three parameters which we'll see in every grid service we program.

Step 4: Create a GAR file with Ant

At this point we have (1) a service interface in GWSDL, (2) a service implementation in Java, and (3) a deployment descriptor telling the grid 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 grid services container? Do we have to copy these files to strategically located directories? And what about the Java file? We haven't compiled it 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'll generate a *Grid Archive*, or *GAR file*. This GAR file is a

single file which contains all the files and information the grid services container need to *deploy* our service and make it available to the whole world. In fact, in the next page we'll instruct a simple grid services container to take the GAR and deploy it.

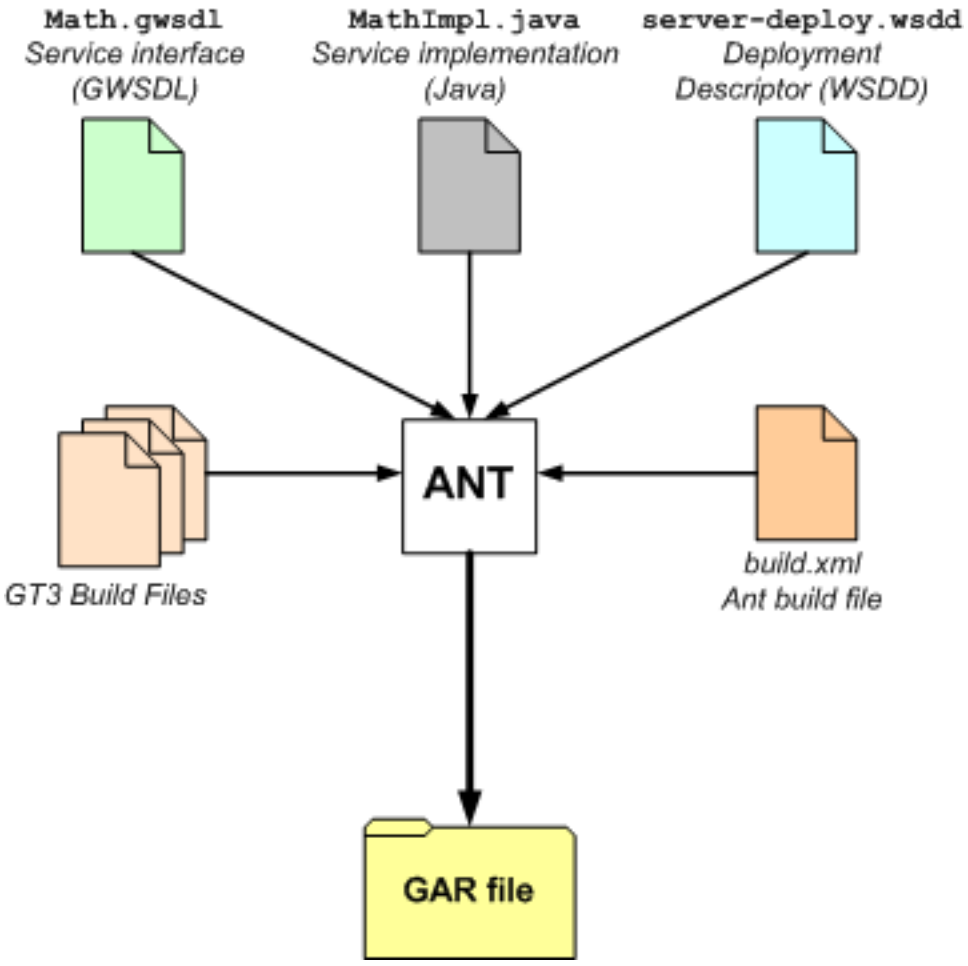
However, creating a GAR file is a pretty complex task which involves the following:

- Converting the GWSDL into WSDL
- 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 *build file* ('Makefile' in the make jargon). This build file 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:



As you can see, Ant generates the GAR directly from the three source files. Internally, it is carrying out all the steps listed earlier, sparing us the cumbersome task of doing them ourselves. In a GT3 project, Ant uses two sets of build files: a couple of build files which are a part of GT3, and a build file we'll have to write on our own. The GT3 build files cover all the important steps (generating the WSDL code, generating the stubs, ...). Our build file essentially has all the unique parameters of our Grid Service, and a bunch of calls to the GT3 build files. At first, it is safe to know practically nothing about Ant and build files; you can usually write a 'generic' build file which will work with more than one Grid Service, and then you won't have to see build files ever again. In fact, this tutorial includes a handy build file that works with all the examples we'll see. However, as you move on to more complex projects, you'll probably need to write custom build files to fine tune the whole build process.

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.

Our handy multipurpose buildfile and script

For the rest of the tutorial, we are going to use a very handy Ant build file which will work with all the examples we'll see. That way, we won't have to rewrite the build file each time we get to a new example. The build file is included in the downloadable tutorial files (available in the tutorial website [http://www.casa-sotomayor.net/gt3-tutorial/]). Since this tutorial isn't meant as an Ant tutorial, we won't see what's inside the build file, but feel free to take a look inside.

Since using the Ant file directly implies passing a lot of parameters to Ant, we'll also use a handy shell script which makes things even simpler (also included with the tutorial files).

However, before doing anything, make sure you create a file called `build.properties` in `$TUTORIAL_DIR` with the following line:

```
ogsa.root=path to GT3 installation
```

Replace *path to GT3 installation* with the path where you installed GT3. For example:

```
ogsa.root=/usr/local/gt3
```

Creating the MathService GAR

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

```
./tutorial_build.sh <service base directory> <service's GWSDL file>
```

The "service base directory" is the directory where we placed the `server-deploy.wsdd` file, and where the MathImpl.java file can be found (inside an `impl` directory). More details on this in the Tutorial Directory Structure appendix.

For example, to build our first example and generate its GAR file, we simply need to do the following:

```
./tutorial_build.sh \
org/globus/progtutorial/services/core/first \
schema/progtutorial/MathService/Math.gwsdl
```

Note

Make sure you run this from `$TUTORIAL_DIR`.

Note

Windows users can use a Python build script originally contributed by Michael Schneider. This script is called `tutorial_build.py` and is included with the downloadable tutorial files. If you prefer to use the Python script, simply replace `tutorial_build.sh` with `tutorial_build.py` in all the following examples.

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

```
$TUTORIAL_DIR/build/lib/org_globus_progtutorial_services_core_first.gar
```

Step 5: Deploy the service into a grid services container

The GAR file, as mentioned in the previous page, contains all the files and information the web server needs to deploy the Grid Service. Deployment is also done with the Ant tool, which unpacks the GAR file and copies the files within (WSDL, compiled stubs, compiled implementation, WSDD) into key locations in the GT3 directory tree. It also reads our deployment descriptor and configures the web server to take our new Grid Service into account.

This deployment command must be run from the root of your GT3 installation. Furthermore, you need to run it with a user that has write permission in that directory.

```
ant deploy -Dgar.name=<full path of GAR file>
```

For the GAR file we've just created, this would be:

```
ant deploy \
-Dgar.name=$TUTORIAL_DIR/build/lib/org_globus_progtutorial_services_core_first.gar
```

Deployment is really as simple as that! That also concludes the five steps necessary to write and deploy a Grid service. However, although you're probably beaming with pride because you've deployed your first Grid 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 Grid Service with a command-line client which will invoke the `add` method and the `getValue` method. This client will receive two arguments from the command line:

1. The Grid Service Handle (GSH)
2. Number to add

The client class will be called `Client` and we'll place it in the `$TUTORIAL_DIR/org/globus/progtutorial/clients/MathService/Client.java` file. Again, you can find more information about the directory structured followed in the tutorial in the Tutorial Directory Structure appendix.

The full code for the client is the following:

```
package org.globus.progtutorial.clients.MathService;

import org.globus.progtutorial.stubs.MathService.service.MathServiceGridLocator;
import org.globus.progtutorial.stubs.MathService.MathPortType;

import java.net.URL;

public class Client
{
```

```
  public static void main(String[] args)
  {
    try
    {
      // Get command-line arguments
      URL GSH = new java.net.URL(args[0]);
      int a = Integer.parseInt(args[1]);


      // Get a reference to the MathService instance
      MathServiceGridLocator mathServiceLocator = new MathServiceGridLocator();
      MathPortType math = mathServiceLocator.getMathServicePort(GSH);

      // Call remote method 'add'
      math.add(a);
      System.out.println("Added " + a);

      // Get current value through remote method 'getValue'
      int value = math.getValue();
      System.out.println("Current value: " + value);
    }catch(Exception e)
    {
      System.out.println("ERROR!");
      e.printStackTrace();
    }
  }
}
```

Note

This file is `$TUTORIAL_DIR/org/globus/progtutorial/clients/MathService/Client.java`

As you can see, writing a Grid Service client is very easy. With only two lines we obtain a reference to the Math portType. In following sections, as we introduce things like service data, notifications, and factories the code to obtain that reference will be slightly longer. However, the important point is that, once we have that reference, we can work with the Grid Service *as if it were a local object*. Notice, however, that we have to put the whole code inside a try/catch block, because the Grid Service methods (in this example, the add method) can throw RemoteExceptions.

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` 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/classes/:$CLASSPATH \
org/globus/progtutorial/clients/MathService/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 MathServiceGridLocator. Before running it, we need to to start up the standalone container. Otherwise, our Grid Service won't be available, and the client will crash. The following command must be run from the root of your GT3 installation:

```
globus-start-container
```

Note

You need to setup the GT3 command-line clients for this command to work. If you have no idea what I'm talking about, take a look at the following section of the How to... appendix: How to setup the GT3 scripts.

When the container starts up, you'll see a list with the GSH 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/ogsa/services/progtutorial/core/first/MathService
```

This is the service as it would appear in a default GT3 installation, with the standalone container located in http://localhost:8080/ogsa/services. The GSH 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/classes/:$CLASSPATH \
org.globus.progtutorial.clients.MathService.Client \
http://127.0.0.1:8080/ogsa/services/progtutorial/core/first/MathService \
5
```

If all goes well, you should see the following:

```
Added 5
Current value: 5
```

This means we've successfully added 5 to the internal value of the service (which is initially zero). To see how Grid Services are truly stateful, you can run the client again. You should see the following:

```
Added 5
Current value: 10
```

This, of course, means that after adding 5 to the internal value, which was previously equal to 5, the internal value is now equal to 10.

Voila! You are now one step closer to the Grid Service Nirvana! :-)

Chapter 4. Operation Providers

In the previous chapter, when implementing our Grid Service we had to create a Java class which extended from GridServiceImpl:

```
public class MathImpl extends GridServiceImpl implements MathPortType
```

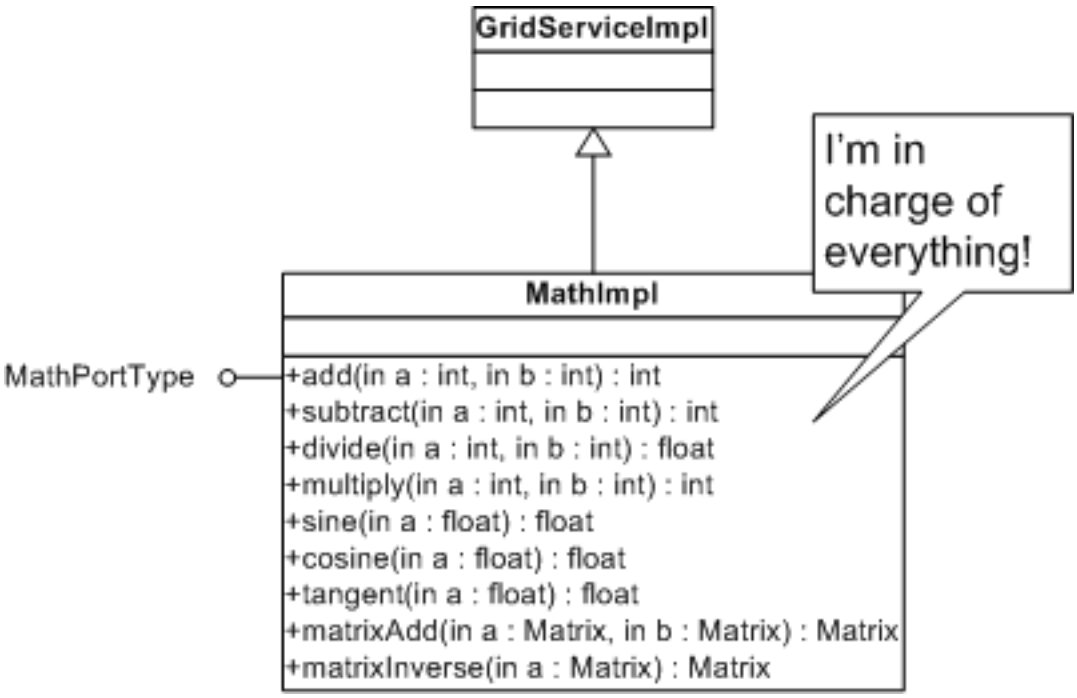
This is an implementation approach known as *implementation by inheritance*, because we get all the basic functionality of a Grid Service from a base class. The Globus Toolkit 3 offers another approach: *implementation by delegation* (or *Operation Providers*, in the GT3 jargon). If you're familiar with object-oriented design patterns, you might already know about this approach (also, if you've worked with CORBA, this is the approach used in the CORBA Tie mechanism). If you have no idea what 'implementation by delegation' is, don't worry: we'll talk about it briefly in the next page.

Inheritance versus Operation Providers

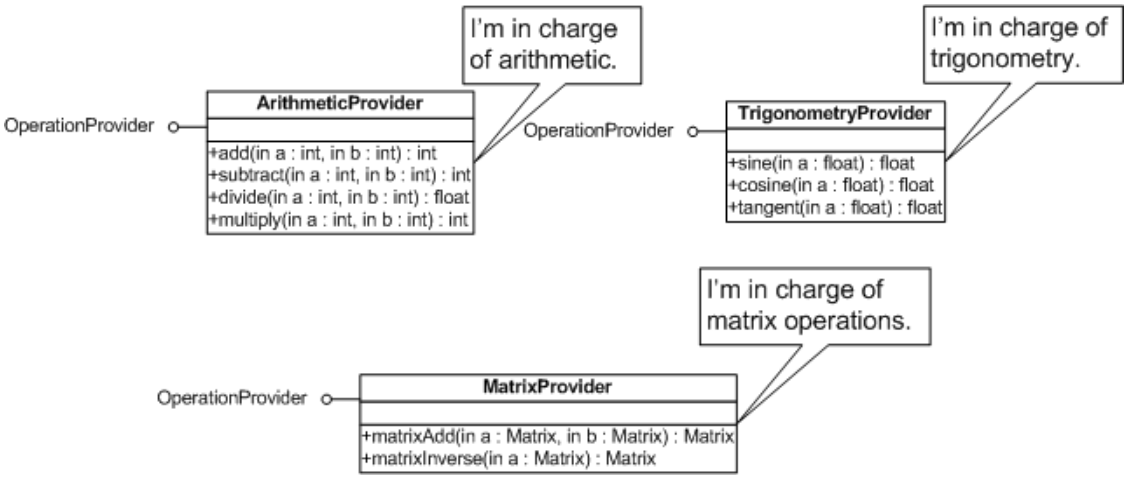
The Globus Toolkit 3 provides two implementation approaches:

- **Implementation by inheritance**
- **Implementation by delegation** (Operation Providers)

The inheritance approach is the one used in the previous chapter. The class that implements the MathService interface, MathImpl, extends from a class called GridServiceImpl. Since GridServiceImpl contains all the basic functionality of a Grid Service, our MathImpl class simply has to provide all the operations specified in our particular portType (in our case, MathPortType):



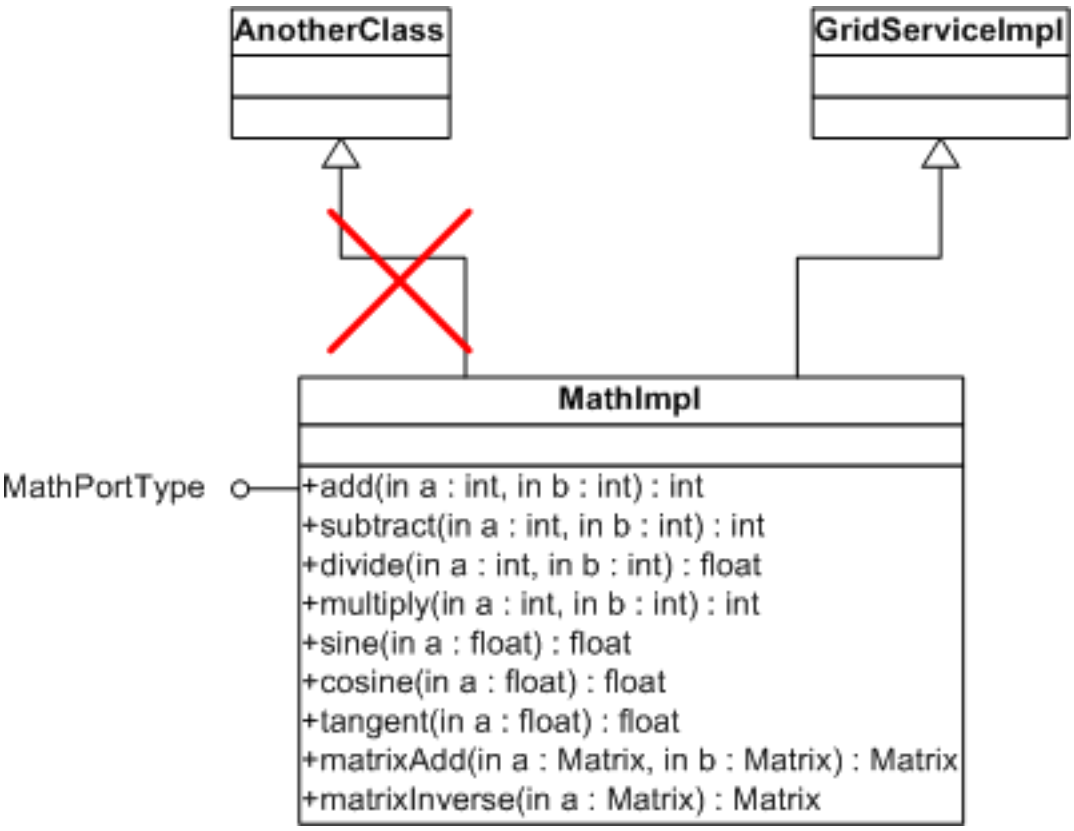
The delegation approach, on the other hand, allows us to distribute the operations of our PortType in several classes. For example, it might make sense to separate the previous MathImpl into three classes: one for the arithmetic operations, one for the trigonometry operations, and one for the matrix operations. Each of these classes is called an *operation provider*.



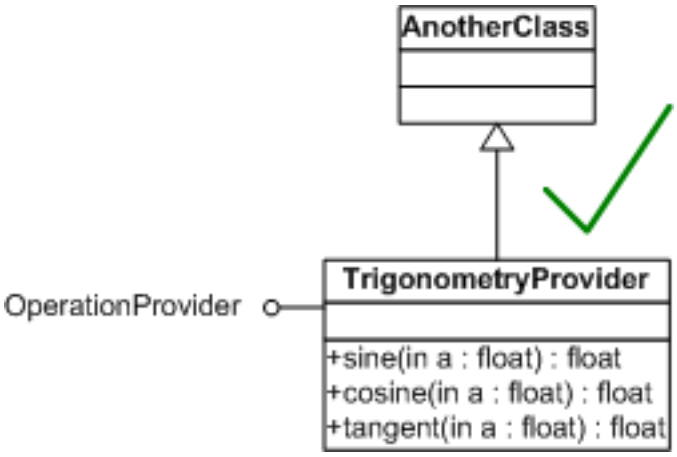
Notice how these classes don't extend from any base class. They only have to implement an interface called OperationProvider. Of course, you might be wondering: If there is no base class to extend from, where do these classes get their 'basic Grid Service functionality'? Actually, GridServiceImpl is still present in this approach. The only difference is we're not forced to extend from it. We'll just need to tell the Grid Service container that GridServiceImpl will supply the basic functionality (this is done with the deployment descriptor; we'll see that in the next section).

So, why is the delegation approach better that the inheritance approach? First of all, imagine you want to write and deploy a Grid Service which has to extend from a Java class you already have. With the inher-

itance approach, you couldn't do this, because your Grid Service already has to extend from GridServiceImpl, And, of course, multiple inheritance is not allowed in Java (and, even if it were, multiple inheritance is considered very bad design).



Using the delegation approach, your provider class is free to extend from any class.



The delegation approach has other advantages, such as favoring more modular, uncoupled, and reusable designs (you can distribute all your operations into several operation providers, and then reuse providers in different Grid Services).

The only disadvantage of using the delegation approach is that it requires just a little bit more work and

code than the inheritance approach. The deployment descriptor needs a couple of extra lines, and you need to implement the methods in the OperationProvider interface. However, this requires only a small (really small) amount of extra work.

Finally, take into account that in GT3 both approaches *are not* mutually exclusive. In other words, we could easily implement part of our portType in a class that extends from GridServiceImpl and then complete the implementation with operation providers. In fact, we'll see that this is a common technique in GT3, since the toolkit includes a lot of operation providers which we can directly 'plug into' our service to add functionality. For example, later on we'll see that to use notifications in a service we barely have to write any code ourselves. We simply have to 'plug in' the 'notifications operation provider' to add that functionality to our service.

Writing an operation provider

Defining the service interface

For this example we'll be using the the exact same GWSDL file we used in the previous example. We can do this because, as long as the *interface* doesn't change (i.e. as long as we don't add methods, remove methods, add parameters to methods, etc.) we can reuse the GWSDL and the stub classes. Using operation providers instead of extending from GridServiceImpl is an *implementation issue* and, therefore doesn't affect the interface.

Implementing the service

The operation provider class (which we will call MathProvider) is very similar to all the MathImpl classes we saw in the previous example. The main difference is that, in this class, we need to implement the OperationProvider interface. Let's take a look at the whole code, and then take a closer look at the new code:

```
package org.globus.progtutorial.services.core.providers.impl;

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

import java.rmi.RemoteException;
import javax.xml.namespace.QName;

public class MathProvider implements OperationProvider
{

  // Operation provider properties
  private static final String namespace = "http://www.globus.org/namespaces/2004/0

  private static final QName[] operations =
    new QName[]
      {
      new QName(namespace, "add"),
      new QName(namespace, "subtract"),
      new QName(namespace, "getValue")
      };

  private GridServiceBase base;

  // Operation Provider methods
  public void initialize(GridServiceBase base) throws GridServiceException
```

```
  {
    this.base = base;
  }

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


  private int value = 0;

  public void add(int a) throws RemoteException
  {
    value = value + a;
  }

  public void subtract(int a) throws RemoteException
  {
    value = value - a;
  }

  public int getValue() throws RemoteException
  {
    return value;
  }
}
```

Note

This file is `$TUTORIAL_DIR/org/globus/progtutorial/services/core/providers/impl/MathProvider.java`

As mentioned before, this class implements the OperationProvider interface, and doesn't extend from any base class:

```
public class MathProvider implements OperationProvider
```

Next, we need to implement the two methods in the OperationProvider interface: `initialize()` and `getOperations()`. These two methods will require three private attributes: `namespace`, `operations`, and `base`.

```
// Operation provider properties
private static final String namespace = "http://www.globus.org/namespaces/2004/02/

private static final QName[] operations =
  new QName[]
    {
    new QName(namespace, "add"),
    new QName(namespace, "subtract"),
    new QName(namespace, "getValue")
    };
```

```
private GridServiceBase base;



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

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

These two methods and three attributes are basically what make the operation provider fit into the Grid Service. The `operations` attribute specifies what operations this class *provides*. The Grid Services container uses the getOperations() method to 'ask' our class what operations it provides.

This list of operations is an array of *fully qualified names*. This means we have to specify both the *namespace* of the operation and its name. The namespace has to be the *target namespace* specified in the GWSDL file which our operation provider is going to implement.

```
private static final String namespace = "http://www.globus.org/namespaces/2004/02/

private static final QName[] operations =
  new QName[]
    {
    new QName(namespace, "add"),
    new QName(namespace, "subtract"),
    new QName(namespace, "getValue")
    };
```

Note

Remember we're reusing the GWSDL file from the previous chapter. If you take a look at it (`$TUTORIAL_DIR/schema/progtutorial/MathService/Math.gwsdl`) you'll see that the target namespace is indeed `http://www.globus.org/namespaces/2004/02/progtutorial/MathService`.

Besides the `initialize` and `getOperations` methods and the `namespace`, `operations`, and `base` attributes, the rest of the implementation is just like the one seen in the previous chapter:

```
private int value = 0;

public void add(int a) throws RemoteException
{
  value = value + a;
}

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

```
  value = value - a;
}

public int getValue() throws RemoteException
{
  return value;
}
```

Deploying the Grid Service

The deployment descriptor needs some slight changes:

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

  <service name="progtutorial/core/providers/MathService" provider="Handler" style
    <parameter name="name" value="MathService (with Operation Provider)"/>
    <parameter name="schemaPath"
       value="schema/progtutorial/MathService/Math_service.wsdl"/>

    <parameter name="className"
       value="org.globus.progtutorial.stubs.MathService.MathPortType"/>
    <parameter name="baseClassName"
       value="org.globus.ogsa.impl.ogsi.GridServiceImpl"/>
    <parameter name="operationProviders"
       value="org.globus.progtutorial.services.core.providers.impl.MathProvider"/>

    <!-- Start common parameters -->
    <parameter name="allowedMethods" value="*"/>
    <parameter name="persistent" value="true"/>
    <parameter name="handlerClass"
       value="org.globus.ogsa.handlers.RPCURIProvider"/>
  </service>
</deployment>
```

Note

This file is `$TUTORIAL_DIR/org/globus/progtutorial/services/core/providers/server-deploy.wsdd`

Notice how the baseClassName is no longer MathImpl, or any class programmed by us (such as MathProvider).

```
<parameter name="baseClassName" value="org.globus.ogsa.impl.ogsi.GridServiceImpl"/
```

Now we're using this parameter to tell the Grid Services container that GridServiceImpl will provide the basic functionality of our Grid Service. The implementation of the methods in our Math PortType will be found in an operation provider, which we specify in the operationProviders parameter:

```
<parameter name="operationProviders"
  value="org.globus.progtutorial.services.core.providers.impl.MathProvider"/>
```

Finally, we need to specify the PortType of our GridService (remember that the operation providers don't have an "implements MathPortType", so the Grid Services container can't use the operation providers to figure out what the PortType is).

```
<parameter name="className" value="org.globus.progtutorial.stubs.MathService.MathP
```

Now, let's build the Grid Service:

```
./tutorial_build.sh \
org/globus/progtutorial/services/core/providers \
schema/progtutorial/MathService/Math.gwsdl
```

Note

Notice how we're reusing the GWSDL file used in the previous chapter.

Finally, let's deploy it and start the Grid Services container:

```
ant deploy \
-Dgar.name=$TUTORIAL_DIR/build/lib/org_globus_progtutorial_services_core_providers

globus-start-container
```

Note

Remember to run these commands from $GLOBUS_LOCATION with a user with write permissions in that directory.

A simple client

Now comes a part which might come as a surprise... to try out the service we won't need to program a new client application. We can reuse the client from the previous section. This is due to the fact that this example and the previous *share the same service interface* (remember how we reused the same GWSDL file). A client is bound to a particular *interface*, not to a particular implementation. So, as long as we don't modify the service interface, we can reuse both the GWSDL file *and* the client. Of course, now we'll tell the client to access a new GSH (the one where we've placed the current operation providers example).

First of all, if you haven't compiled the previous example's client (or have erased the compiled class), be sure to compile it now:

```
javac \
-classpath ./build/classes/:$CLASSPATH \
```

```
org/globus/progtutorial/clients/MathService/Client.java
```

Now, run the client:

```
java \
-classpath ./build/classes/:$CLASSPATH \
org.globus.progtutorial.clients.MathService.Client \
http://127.0.0.1:8080/ogsa/services/progtutorial/core/providers/MathService \
5
```

The output should be *exactly the same* as the previous example's output. Remember that the only thing that's changed from the previous example to this one is that we're using operation providers to implement the service, instead of extending from GridServiceImpl. We haven't modified the internal logic of the service, so the output should be the following:

```
Added 5
Current value: 5
```

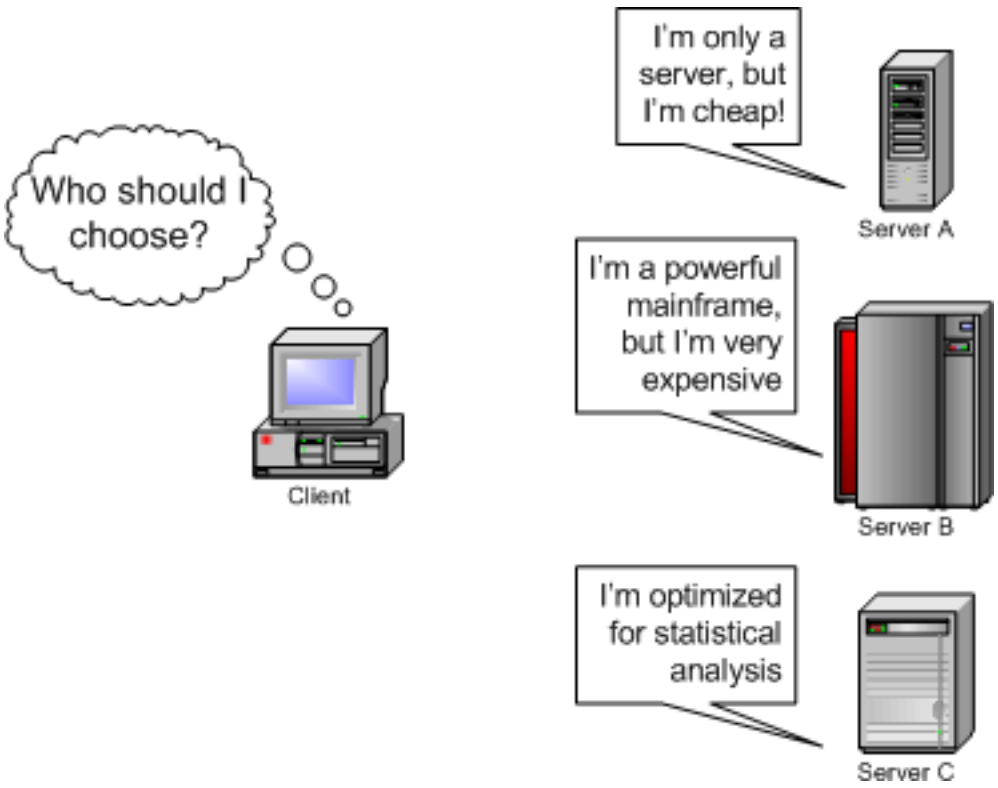
Chapter 5. Service Data

Service Data is one of the main improvements Grid Services introduce with respect to plain Web Services. The concept of 'service data' was briefly introduced in the What is a Grid Service? page. In this section we will explore the concept of service data further (in a general sense), and then we'll see how we can include service data in our grid services.

The logic behind Service Data

In A short introduction to Web Services we saw that one of the important parts of the Web Services architecture is *service discovery*, which allows us to find out the URI of a Web Service which meets our requirements. We mentioned one example: we might need a Web Service that is capable of giving us the temperature in US cities. We saw that a UDDI registry would help us find those Web Services, but there was something we didn't mention: How exactly does a Web Service *advertise* what kind of services it offers? Your first guess might be WSDL, since it is the description language of the Web Services architecture. However, WSDL is too 'technical'. It is concerned with details such as method invocation, protocols, etc. We need something which is easier to classify and index. This is what we can achieve with *Service Data*. Although Service Data can be handled in different ways in standard Web Services, in this section we will only see the solution proposed in the OGSI specification (i.e. in Grid Services).

Service Data is a structured collection of information that is associated to a Grid Service. This information must be easy to query, so Grid Services can be classified and indexed according to their characteristics. For example, we might have a client that needs to use a MathService to perform a mind-boggling calculation. We might have many different MathServices in our organization, and the client needs to know which one can satisfy its needs.



The client can choose thanks to the Service Data associated to each MathService. If the client needs the

most powerful service, without any consideration to how much it will cost, then it will probably choose the mainframe. If the decisive factor is cost, then Server A will probably be the best choice. However, if the client wants to do some sort of statistic analysis, then Server C is the best bet. Of course, this is an over-simplification: usually the client won't implement some complicated algorithm to query Service Data and choose a service. It will probably delegate that task on an *index service*, which will make the decision based on multiple criteria: speed, cost, availability, etc. The criteria are usually specific to each particular problem; in the 'US city temperature' example, the criteria might be 'number of cities available', 'availability of temperature, humidity, pressure, etc.', 'frequency of updates', etc.

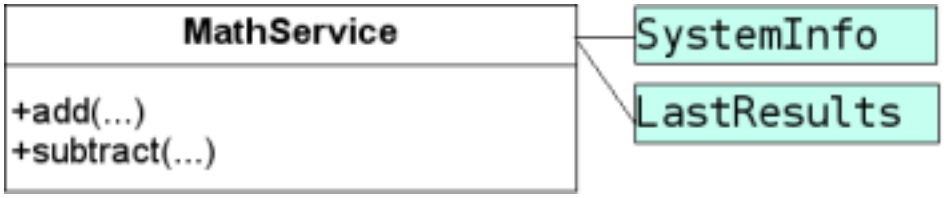
Summing up, Service Data is an essential part of service discovery. However, we will also see in the next section that Service Data and notifications are closely related.

Service Data in Grid Services

Any Grid Service can have any number of *Service Data Elements* (or SDEs) associated to it. Each SDE must have a name that is unique within that particular type of Grid Service. An SDE contains structured data (internally represented in XML). The structure of this data (e.g. is it a fundamental data type like an integer? is it an array? is it a class with a set of attributes?) is defined by the programmer in the GWSDL file.

A simple example

All this stuff about SDEs and structured data might seem a bit confusing, so let's take a look at a simple example. We could define two SDEs in our MathService:



An SDE can, in turn, have any number of *values*. Don't worry if this concept of 'value of an SDE' (or, rather, value**s** of an SDE) is a bit confusing. We'll clear it up by looking into the two SDEs we've just defined: `SystemInfo` and `LastResults`.

The **SystemInfo** SDE could be used to store information about the system where MathService is running. Without looking at the exact mechanism with which we can define the structure of the SDE, we can say that the datatype of the SystemInfo SDE is a structure with the following fields:

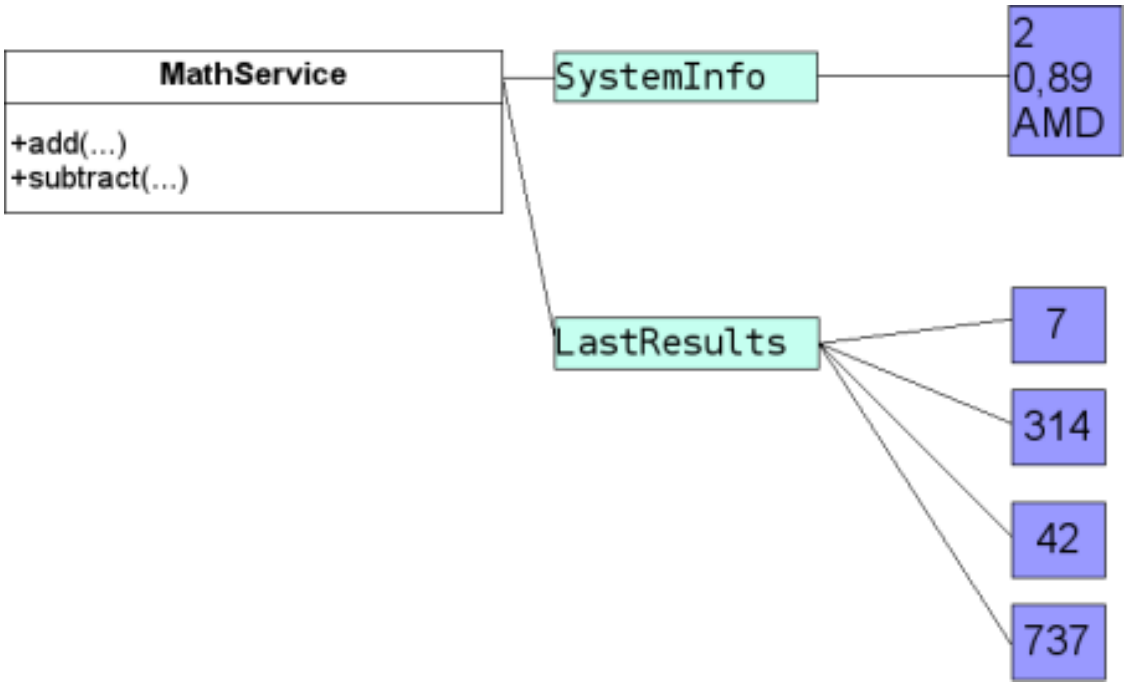
- numberOfCPUs: integer
- systemLoad: float
- CPUBrand: string

Of course, it only makes sense to have one (and only one) of these SDEs attached to our service. Therefore, we can specify (in the GWSDL file) that this particular SDE will have a *cardinality* of 1..1 (minimum quantity of one, maximum quantity of one).

The **LastResults** SDE, on the other hand, could be used to store the last *N* internal values that have been stored in MathService. Since the internal value is an integer, we could simply define the datatype of the `LastResults` SDE as 'integer'. If we wanted to keep (for example) the last 10 values, we would define the cardinality of `LastResults` as 0..10 (minimum quantity of zero, maximum quantity of

ten). Of course, we could've also chosen to define the datatype as 'array[10] of integer' and set the SDEs cardinality as 1..1. That would also be a possible solution, but let's stick with the 0..10 cardinality since it shows how an SDE can have more than on value.

Summing up, our MathService could look like this at a given point in its lifetime:

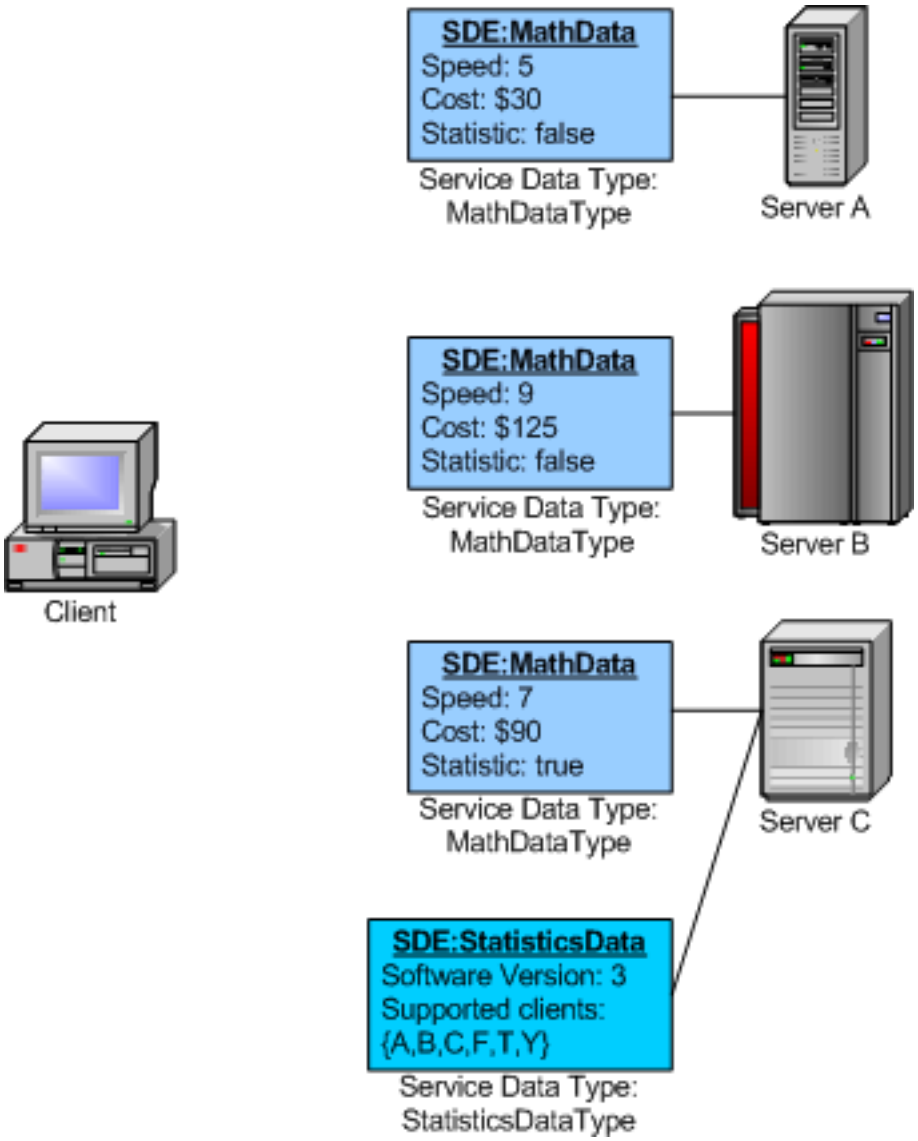


Notice how `SystemInfo` has one value (and how that single value is a complex datatype: it has three attributes) and how `LastResults` has *four* values (which is perfectly valid since the cardinality is 0..10), all with the same datatype (integer).

Finally, remember from the What is a Grid Service? page that service data generally falls into two categories: *state information* (operation results, intermediate results, runtime information, etc.) and *service metadata* (system data, supported interfaces, cost of using the service, etc.). The `SystemInfo` SDE would fall into the 'service metadata' category, while the `LastResults` SDE would fall into the 'state information' category.

A slightly less simple example

Looking back at the diagram shown in the previous page (the diagram with the client that had to choose one of three servers), we could represent the information each server exposed as SDEs like this:



In this diagram, we have three MathServices. The first two services each have one MathData SDE. Notice how both SDEs contain the same *type* of information (MathDataType, containing speed, cost, and statistics). The third service has two SDEs: a MathData and a StatisticsData one. Notice how the StatisticsData SDE has a different type of information (StatisticsDataType, containing software version and supported clients)

Imagine our client needed a MathService to perform a calculation, and that it is imperative to perform it as quickly as possible. The client would ask each MathService for its MathData SDE, then it would look at the speed property, and then choose the service with the highest speed. If our client needed to perform statistical analysis, it would choose the third instance (the only one with the "statistics" property set to true). Then, it would ask for the StatisticsData SDE to make sure details like the software version and the supported clients are acceptable.

Again, remember that a client usually doesn't have to mess around with these kind of algorithms to decide what service is better. This task is usually handled by an index service.

So... where and how exactly do we define Service Data?

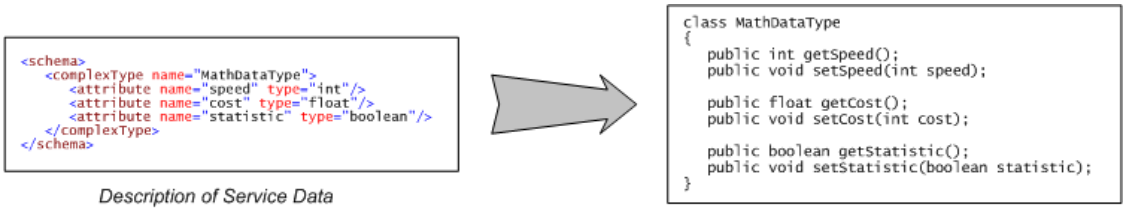
Ok, you'll agree that Service Data is definitely a good idea. However, how exactly do we implement Service Data? For example, should we simply add 'speed', 'cost', and 'statistics' attributes to the MathService implementation and expose them through get/set methods? Well, this *could* be a way to expose data about our service, but it's not a very elegant solution. What about having more than one SDE? What about not having any SDEs?

The solution is in the GWSDL. As we'll see in the next page, another of the extensions introduced by GWSDL (with respect to WSDL) is that we can *associate* SDEs to a portType, specifying the cardinality of each SDE along with other properties. The datatype of each SDE is specified in XML Schema [http://www.w3c.org/XML/Schema], a language originally intended to describe the structure and vocabulary of XML documents. However, it can also be used to define other types of structures (including databases, objects, etc.)

For example, let's take the `LastResults` SDE, which has an integer datatype (a fundamental datatype). For this SDE we would we simply have to specify (in GWSDL) that its datatype is `xsd:int`.

The SystemInfo SDE, on the other hand, has a *complex datatype*. In other words, it is a structure with a set of attributes (or fields). To specify that we want that SDE to be of that datatype, we must first define that custom structure. This is also done in XML Schema, either directly in the <types> element of the GWSDL files or in a separate XML Schema file (XSD extension) which is imported into the GWSDL file. In fact, in the example we're going to implement we'll take the latter approach.

In the case of SDEs with complex datatypes, a Java Bean is created so we can interact with that SDE from Java.



Description of Service Data Element in XML Schema

Java Bean

Note

This code is not 100% correct...it has been shortened for simplicity.

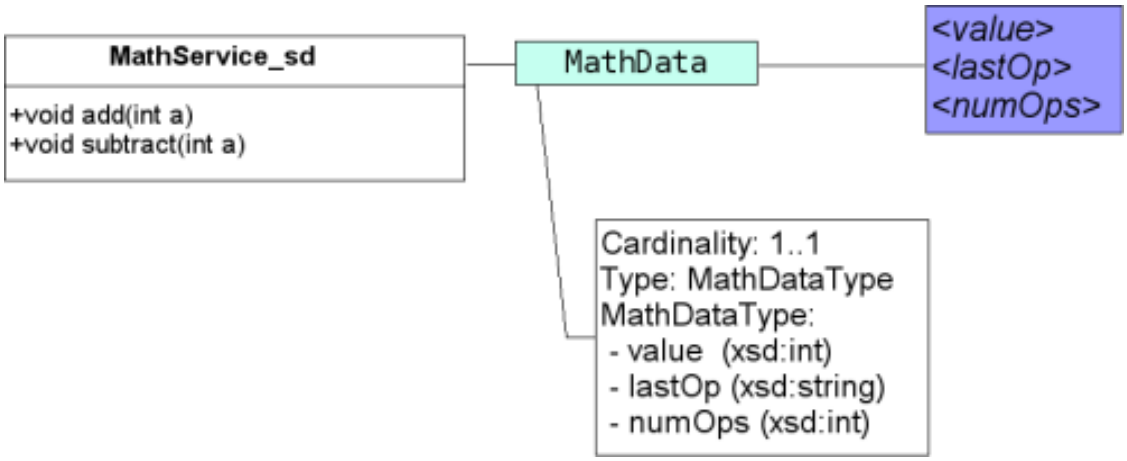
Accessing the SDEs, as we'll see in the next page, is very easy. The standard GridService portType includes a method which allows us to tell the service "Hey, you, give me the SDE called FOOBAR!". First of all, if the SDE is *multivalued* (like the LastResults SDE), we'll receive an array of SDEs (otherwise, we'd receive an individual SDE). Then, if the SDE has a fundamental datatype (integer, float, string, etc.) we'll get that fundamental datatype directly when we access the value of the SDE. If the SDE has a complex datatype, we'll receive an 'SDE Java Bean' which we can use to access the individual fields of the SDE.

A service with Service Data

We're now going to modify the MathService example to add some Service Data. Since some of the files are a bit long, we won't show the whole code in this page, only the important parts. For the complete code, please download the tutorial files in the tutorial website [http://www.casa-sotomayor.net/gt3-tutorial].

The MathData SDE

We are going to add a single SDE to MathService:



This SDE will have one and only one value (cardinality 1..1). The datatype of this SDE will be a complex datatype called MathDataType, which will contain the internal value of MathService, the name of the last operation invoked (addition or subtraction), and the number of times either of the operations has been invoked. Although we could define MathDataType in the GWSDL file, we'll define it instead in a separate XML Schema file called MathDataType.xsd:

```
<complexType name="MathDataType">
  <sequence>
    <element name="value" type="int"/>
    <element name="lastOp" type="string"/>
    <element name="numOps" type="int"/>
  </sequence>
</complexType>
```

Note

This is a part of `$TUTORIAL_DIR/schema/progtutorial/MathService_sd/MathSDE.xsd`

If you take a close look at the `MathSDE.xsd` file, you'll see that it isn't actually pure XML Schema. It is a very simple GWSDL description with a <types> tag (no messages, PortTypes or bindings) with just one type: the MathDataType. If you need to create your own SDDs, the only change you would need to make to the file (besides defining your own schema) would be to define a new target namespace.

The Java Bean generated from the XML Schema would look something like this:

```
public class MathDataType  implements java.io.Serializable {
    private int value; // attribute
    private java.lang.String lastOp; // attribute
    private int numOps; // attribute

    public MathDataType() {
```

```
    }

    public int getValue() {
        return value;
    }

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

    public java.lang.String getLastOp() {
        return lastOp;
    }

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

    public int getNumOps() {
        return numOps;
    }

    public void setNumOps(int numOps) {
        this.numOps = numOps;
    }
}
```

Note

This is an *extract* from the code generated from the XML Schema file. This Java Bean is generated in the compile/deploy process by Ant, so **don't** use this code directly.

Service Interface

In this example, we are modifying the service interface, not so much because we'll be changing the specification of the operations (slightly), but mainly because we'll be adding service data. Adding service data to a service is an *interface issue*, so we need to create a new GWSDL file (along with a new client that will be capable of accessing the service data). However, the GWSDL code is going to be very similar to the one from the previous two examples, so it won't be hard to understand. The new GWSDL interface can be found in `$TUTORIAL_DIR/schema/progtutorial/MathService_sd/Math.gwsdl`.

The first change is that we need to specify a new target namespace. Notice how we're adding "`_sd`" after the previous namespace to denote that this is Math Service *with* service data (this is just for clarity, it's not mandated by GWSDL). Then, we need to include two new namespaces. The first corresponds to the SDE type (if you take a look at MathSDE.xsd, you'll see what the target namespace of the SDE type is). The second namespace is an OGSI namespace which contains a set of service data-related definitions.

```
<definitions name="MathService"
  targetNamespace="http://www.globus.org/namespaces/2004/02/progtutorial/MathServi
  xmlns:tns="http://www.globus.org/namespaces/2004/02/progtutorial/MathService_sd"
  xmlns:data="http://www.globus.org/namespaces/2004/02/progtutorial/MathService_sd
  xmlns:ogsi="http://www.gridforum.org/namespaces/2003/03/OGSI"
  xmlns:gwsdl="http://www.gridforum.org/namespaces/2003/03/gridWSDLExtensions"
  xmlns:sd="http://www.gridforum.org/namespaces/2003/03/serviceData"
```

```
  xmlns:xsd="http://www.w3.org/2001/XMLSchema"
  xmlns="http://schemas.xmlsoap.org/wsdl/">
```

Next, we'll need to import the schema file with the description of our MathDataType. Once again, notice how the namespace must correspond with the target namespace in MathSDE.xsd.

```
<import location="MathSDE.xsd"
  namespace="http://www.globus.org/namespaces/2004/02/progtutorial/MathService_sd/
```

Finally, we have to add a new tag (from the service data namespace) inside the <gwsdl:portType> tag. This new tag is <sd:serviceData>, and allows us to specify the properties of each of the SDEs in this Grid Service.

```
<gwsdl:portType name="MathPortType" extends="ogsi:GridService">

  <!-- <operation>s -->

  <sd:serviceData name="MathData"
    type="data:MathDataType"
    minOccurs="1"
    maxOccurs="1"
    mutability="mutable"
    modifiable="false"
    nillable="false">
  </sd:serviceData>
</gwsdl:portType>
```

The first attribute is the name of the SDE. Remember, the name of an SDE must be locally unique (unique within this particular Grid Service). The following attribute specifies the type of this SDE, which is specified in `MathSDE.xsd` (notice how we're using the `data` namespace, which corresponds with the target namespace of `MathSDE.xsd`).

The following attributes refer to various properties of the SDE:

- **minOccurs** : The minimum number of values that this SDE can have.
- **maxOccurs** : The maximum number of values that this SDE can have. The value of this attribute can be unbounded, which indicates an array with no size limit.
- **modifiable** : True or false. Specifies if the value of this SDE can be changed by a client.
- **nillable** : True or false. Specifies if the value of this SDE can be NULL.
- **mutability** : This attribute can have the following values:
  - static: The value of the SDE is provided in the GWSDL description.
  - constant: The value of the SDE is set when the Grid Service is created, but remains constant after that.

- extendable: New elements can be added to the SDE, but not removed.
- mutable: New elements can be added and removed.

In our example, the MathData SDE will have one and only one value (i.e. cardinality 1..1), will be allowed to change during the lifetime of the Grid Service, but cannot be NULL or be modified by the client.

Finally, since the internal value (which is modified by `add` and `subtract`) is now in the MathData SDE, there is no more need to keep the `getValue` method, since we'll be accessing the value through the SDE. If you take a look at the GWSDL file, you'll notice that `getValue` has been eliminated.

Namespace mappings

Since we have a new interface, with a new target namespace, we need to map that namespace to Java packages so the stubs are generated correctly. Notice how we also have to add a mapping for the MathDataType target namespace.

```
http\://www.globus.org/namespaces/2004/02/progtutorial/MathService_sd=
  org.globus.progtutorial.stubs.MathService_sd

http\://www.globus.org/namespaces/2004/02/progtutorial/MathService_sd/bindings=
  org.globus.progtutorial.stubs.MathService_sd.bindings

http\://www.globus.org/namespaces/2004/02/progtutorial/MathService_sd/service=
  org.globus.progtutorial.stubs.MathService_sd.service

http\://www.globus.org/namespaces/2004/02/progtutorial/MathService_sd/MathSDE=
  org.globus.progtutorial.stubs.MathService_sd.servicedata
```

Note

These lines can be found in $TUTORIAL_DIR/namespace2package.mappings

Service Implementation

There's also a lot of new things in the service implementation `$TUTORIAL_DIR/org/globus/progtutorial/services/core/servicedata/impl/` (`MathImpl.java`). The class declaration, however, is still the same:

```
public class MathImpl extends GridServiceImpl implements MathPortType
```

The class will now have two new private attributes. One is the SDE (ServiceData mathDataSDE) and the other is the *value* of the SDE (MathDataType mathDataValue).

```
private ServiceData mathDataSDE;
private MathDataType mathDataValue;
```

The creation of the SDEs takes place in a special method called the `postCreate` method. This is called a *callback method*, which will be explained in greater detail in the lifecycle section. For now, suffice it to say that if we implement this method, the code in the method will be executed right after the service has been created. This is where we will set the initial value of the SDE.

```
public void postCreate(GridContext context) throws GridServiceException
{
  // Call base class's postCreate
  super.postCreate(context);

  // Create Service Data Element
  mathDataSDE = this.getServiceDataSet().create("MathData");

  // Create a MathDataType instance and set initial values
  mathDataValue = new MathDataType();
  mathDataValue.setLastOp("NONE");
  mathDataValue.setNumOps(0);
  mathDataValue.setValue(0);

  // Set the value of the SDE to the MathDataType instance
  mathDataSDE.setValue(mathDataValue);

  // Add SDE to Service Data Set
  this.getServiceDataSet().add(mathDataSDE);
}
```

The steps we must follow to create an SDE and add it to the Service Data Set are:

1. Create a new SDE. Notice how we don't create it directly: we have to call the create method of the Service Data Set. This SDE is initially *empty*, it has no value. Also, the name of the SDE will be MathData

2. Set a value for the SDE. The value of the SDE will be a MathDataType that we create ourselves.

3. Set the initial values of MathDataType. In our example, the last operation is "NONE" and the number of operations done is zero.

4. Add the SDE to the Service Data Set.

The add and subtract methods now have to modify the Service Data each time they are called (to update the 'value', the 'last operation' and the 'number of operations done'). We do this using the private mathDataValue attribute, and an extra method called incrementOps.

```
public void add(int a) throws RemoteException
{
  mathDataValue.setLastOp("Addition");
  incrementOps();
  mathDataValue.setValue(mathDataValue.getValue() + a);
}
```

The incrementOps method is a simple private method that increments the number of operations (in the Service Data) by one.

```
// This method updates the MathData SDE increasing the
// number of operations by one
private void incrementOps()
{
  int numOps = mathDataValue.getNumOps();
  mathDataValue.setNumOps(numOps + 1);
}
```

Deployment Descriptor

The deployment descriptor barely changes: new class, new baseClass, and new schemaPath (all corresponding to the files we've just created).

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

  <service name="progtutorial/core/servicedata/MathService" provider="Handler" sty
    <parameter name="name" value="MathService"/>

    <parameter name="baseClassName" value="org.globus.progtutorial.services.core.s
    <parameter name="className" value="org.globus.progtutorial.stubs.MathService_s
    <parameter name="schemaPath" value="schema/progtutorial/MathService_sd/Math_se

    <!-- Start common parameters -->
    <parameter name="allowedMethods" value="*"/>
    <parameter name="persistent" value="true"/>
    <parameter name="handlerClass" value="org.globus.ogsa.handlers.RPCURIProvider"
  </service>

</deployment>
```

Note

This file is `$TUTORIAL_DIR/org/globus/progtutorial/services/core/servicedata/server-deploy.wsdd`

Compile and deploy

Let's compile this new service. Notice how we're using the *new* GWSDL file.

```
./tutorial_build.sh \
org/globus/progtutorial/services/core/servicedata \
schema/progtutorial/MathService_sd/Math.gwsdl
```

Now, deploy the GAR file and start the service container:

```
ant deploy \
-Dgar.name=$TUTORIAL_DIR/build/lib/org_globus_progtutorial_services_core_serviceda

globus-start-container
```

Remember, you have to run this from your GT3 installation directory, with a user with write permissions on that directory.

A client that accesses Service Data

We will now modify the original MathService client so it will also access the MathData SDE. Instead of just adding two numbers, it will also ask MathService for the MathData SDE. Before performing the addition, it will show us the content of the SDE (the current value, the previous operation, and the number of operations performed). First let's take a look at the complete source code:

```
package org.globus.progtutorial.clients.MathService_sd;

import org.gridforum.ogsi.ExtensibilityType;
import org.gridforum.ogsi.ServiceDataValuesType;
import org.globus.ogsa.utils.AnyHelper;
import org.globus.ogsa.utils.QueryHelper;

import org.globus.progtutorial.stubs.MathService_sd.service.MathServiceGridLocator
import org.globus.progtutorial.stubs.MathService_sd.MathPortType;
import org.globus.progtutorial.stubs.MathService_sd.servicedata.MathDataType;
import java.net.URL;

public class Client
{
  public static void main(String[] args)
  {
    try
    {
      // Get command-line arguments
      URL GSH = new java.net.URL(args[0]);
      int a = Integer.parseInt(args[1]);

      // Get a reference to the Math PortType
      MathServiceGridLocator mathServiceLocator = new MathServiceGridLocator();
      MathPortType math = mathServiceLocator.getMathServicePort(GSH);


      // Get Service Data Element "MathData"
      ExtensibilityType extensibility =
        math.findServiceData(QueryHelper.getNamesQuery("MathData"));
      ServiceDataValuesType serviceData = AnyHelper.getAsServiceDataValues(extensi
      MathDataType mathData =
        (MathDataType) AnyHelper.getAsSingleObject(serviceData, MathDataType.class

      // Write service data
      System.out.println("Value: " + mathData.getValue());
      System.out.println("Previous operation: " + mathData.getLastOp());
      System.out.println("# of operations: " + mathData.getNumOps());

      // Call remote method
      math.add(a);
    }catch(Exception e)
    {
```

```
      System.out.println("ERROR!");
      e.printStackTrace();
    }
  }
}
```

Note

This file is `$TUTORIAL_DIR/org/globus/progtutorial/clients/MathService_sd/Client.java`

The only new code (besides the different package names) is shown in bold. First of all, we need to get the SDE called MathData. We use a GridService method called findServiceData, and a 'helper' class to resolve the name into something the Grid Service can understand. If you're unfamiliar with 'helper' classes, they're very usual in distributed systems programming (if you've used CORBA, you should be familiar with helper classes and the infamous `narrow` method). Just think of 'helper' classes as set of classes that allow us to perform really complicated casting operations.

```
ExtensibilityType extensibility = math.findServiceData(QueryHelper.getNamesQuery("
```

The first thing that probably strikes you as odd from this code snippet is the fact that we're invoking the findServiceData directly on math (which is a MathPortType). Doesn't that portType only have an `add` and a `subtract` method? Yes, that's right, but remember that our Math portType *extends* from a standard portType called GridService. This means we can also use our math portType to invoke GridService methods (such as findServiceData).

Notice how the findServiceData doesn't return a MathDataType class, but an ExtensibilityType class. Now we need to cast it into a MathDataType using more helper classes:

```
ServiceDataValuesType serviceData =
  AnyHelper.getAsServiceDataValues(extensibility);
MathDataType mathData =
  (MathDataType) AnyHelper.getAsSingleObject(serviceData, MathDataType.class);
```

Now that we have a MathDataType object, we can use it like any other local object.

```
System.out.println("Value: " + mathData.getValue());
System.out.println("Previous operation: " + mathData.getLastOp());
System.out.println("# of operations: " + mathData.getNumOps());
```

Compile and run

Let's compile the client:

```
javac \
-classpath ./build/classes/:$CLASSPATH \
org/globus/progtutorial/clients/MathService_sd/Client.java
```

The client receives two parameters:

1. The GSH
2. Number to add

If you run it several times:

```
java \
-classpath ./build/classes/:$CLASSPATH \
org/globus/progtutorial/clients/MathService_sd/Client \
http://127.0.0.1:8080/ogsa/services/progtutorial/core/servicedata/MathService \
5
```

...you will notice how the number of operations increases with each call. However, since we're only performing addition, the 'previous operation' will always be the same.

The GridService Service Data

Besides all the Service Data we might add by ourselves to a Grid Service (such as our MathData), all Grid Services have a set of common Service Data Elements which describe certain characteristics of the Grid Service, such as the GSH of the instance. These SDEs are part of the GridService portType (remember: if we want a portType to be a grid service, it *must* extend from the GridService portType).

We're going to take a brief look at these SDEs "just for the fun of it", since you probably won't need to access them unless you're writing a program that has to dynamically discover low-level data about the Grid Service. However, since we'll also see a small program which displays all the values of these SDEs, it'll give us a chance to see how to deal with more elaborate Service Data scenarios (such as multivalued SDEs and nillable SDEs).

The following are perhaps the most interesting GridService SDEs:

- **gridServiceHandle**. Multivalued SDE which contains the GridService's GSHs.
- **factoryLocator**. Single valued SDE with the locator for the factory which created this Grid Service. If the Grid Service was not created by a factory, the value of this SDE will be null.
- **terminationTime**. Single valued SDE with information about the termination time of the Grid Service.
- **serviceDataNames**. Multivalued SDE with the names of all the SDEs in the Grid Service.
- **interfaces**. Multivalued SDE with the names of all the interfaces (PortTypes) implemented by this Grid Service.

The GridService PortType does have more SDEs (gridServiceReference, findServiceDataExtensibility, and setServiceDataExtensibility) but they have not been included since this is not intended as an ex-

haustive text on GridService SDEs. For more details on the remaining SDEs, take a look at the OGSI specification (you can find a link to it in the Related Documents section of the Introduction).

The PrintGridServiceData client

The examples file (available in the tutorial files available in the tutorial website [http://www.casa-sotomayor.net/gt3-tutorial]) includes a program which, given a GSH, prints all the GridService SDEs mentioned above. You can find the source code at `$TUTORIAL_DIR/org/globus/progtutorial/clients/GridService/PrintGridServiceData.java`. We will not review the code here, but feel free to take a close look at it, to see how multivalued SDEs and nillable SDEs are handled.

We're going to try it out with the Grid Service we've written in this section. Of course, you can try the program with any of the Grid Services we've seen so far. The client receives only one parameter: the Grid Service GSH.

```
javac org/globus/progtutorial/clients/GridService/PrintGridServiceData.java
```

```
java \
org.globus.progtutorial.clients.GridService.PrintGridServiceData \
http://127.0.0.1:8080/ogsa/services/progtutorial/core/servicedata/MathService
```

You should see the following on your terminal:

```
gridServiceHandle: http://127.0.0.1:8080/ogsa/services/progtutorial/core/serviceda
factoryLocator: Not created by a factory!
terminationTime (after): The service plans to exist indefinitely
terminationTime (before): The service has no plans to terminate.
terminationTime (timestamp): Tue Mar 16 18:48:54 CET 2004
serviceDataName: gridServiceHandle
serviceDataName: {http://www.globus.org/namespaces/2004/02/progtutorial/MathServic
serviceDataName: factoryLocator
serviceDataName: serviceDataName
serviceDataName: interface
serviceDataName: setServiceDataExtensibility
serviceDataName: terminationTime
serviceDataName: gridServiceReference
serviceDataName: findServiceDataExtensibility
interface: GridService
interface: {http://www.globus.org/namespaces/2004/02/progtutorial/MathService_sd}M
```

Notice how, since this service hasn't been deployed following the factory/instance model, this Grid Service has no factory locator (`Not created by a factory!`). Also, notice how this Grid Service implements the GridService portType and our own MathPortType portType.

As mentioned above, remember we are doing this just "for the fun of it". Try this client with other Grid Services, and notice how the values vary according to the characteristics of the Grid Service. For example, if you try the client with the notification examples we will see in the following section, you'll see that our Grid Service now implements a new PortType: the NotificationSource PortType.

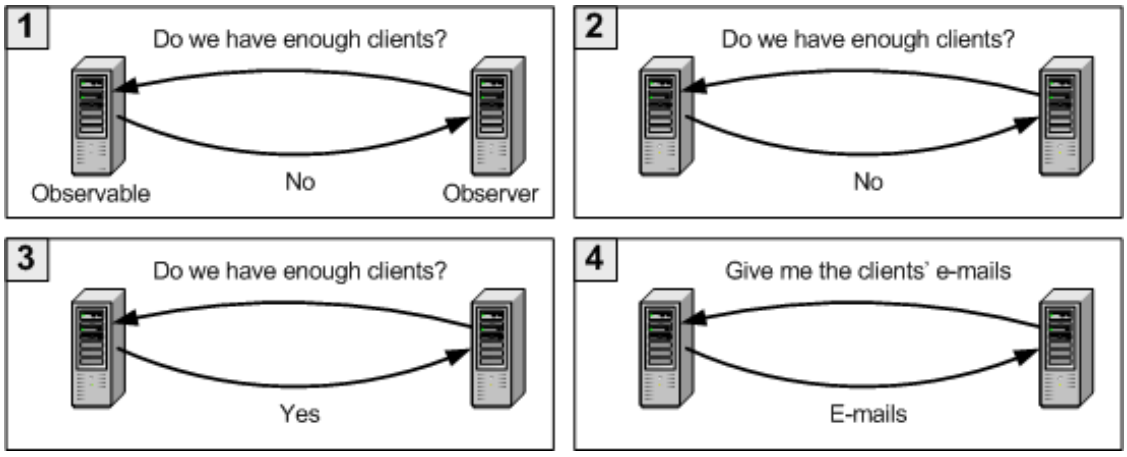
Chapter 6. Notifications

In this section we will see notifications, a core feature which is closely related to service data. Notifications allow clients to be *notified* of changes that occur in a Grid Service. After explaining what notifications are, and how they are approached in GT3, we will see an example service which uses notifications.

What are notifications?

Notifications are nothing new. It's a very popular software design pattern, although you might know it with a different name: Observer/Observable, Model-View-Controller, etc. 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 make a set of specific calls to the server.

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 we have a server where clients can sign up for a special newsletter. The newsletter is sent as soon as a certain number of clients have signed up, but the newsletter e-mails are sent by *another* server. This other server needs to know when enough clients have signed up, so it can send the e-mails. The first server is called the *observable* part, and the second server is called the *observer* part. The polling approach would go like this:



1. The observer asks the observable if there are any changes. The observable replies "No", so the observer waits a while before making another call.

2. Once again, the observer asks the observable if there are any changes. The observable replies "No", so the observer waits a while before making another call.

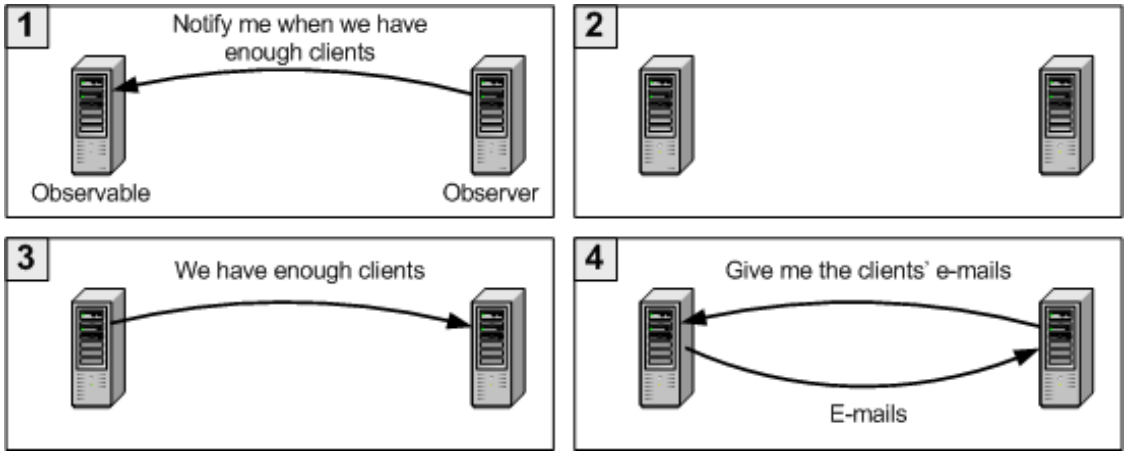
3. Once again, the observer asks the observable if there are any changes. This time the observable replies "Yes".

4. Now that the data is available, the observer asks the observable for the e-mails.

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 observer. If we have dozens of observers, the observable 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 observable if there are any changes, we make an initial call asking the observable to *notify* whenever there are any changes. The observable will contact us as soon as a change occurs, and then we can act accordingly. This is the *notification* approach.



1. The observer asks the server to notify him as soon as there are enough clients. The observable keeps a list of all its registered observers. This step is normally called the subscription or registration step.
2. The observer and the observable go about their business. So far, there aren't enough clients.
3. Enough clients have subscribed. The observable *notifies* all its observers (remember, there can be more than one) that there are enough clients.
4. The observer asks the observable to send the e-mail addresses.

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).

Pull Notifications vs. Push Notifications

There are two ways of applying the Observer/Observable design pattern: the pull approach or the push approach. Each approach has its advantages and its disadvantages.

- **Pull**: The pull approach is the one shown in the above diagrams. In a strict implementation, the observable will simply tell the observers that 'a change' has occurred. The notification can specify what type of change it is (e.g. "We have enough clients") but will never include the information relative to that change. Notice how, after the notification, the observer must make another call to the observable to get the e-mails. This might seem like a redundant call (we could have sent the e-mails along with the notification), but this approach is useful in the following cases:

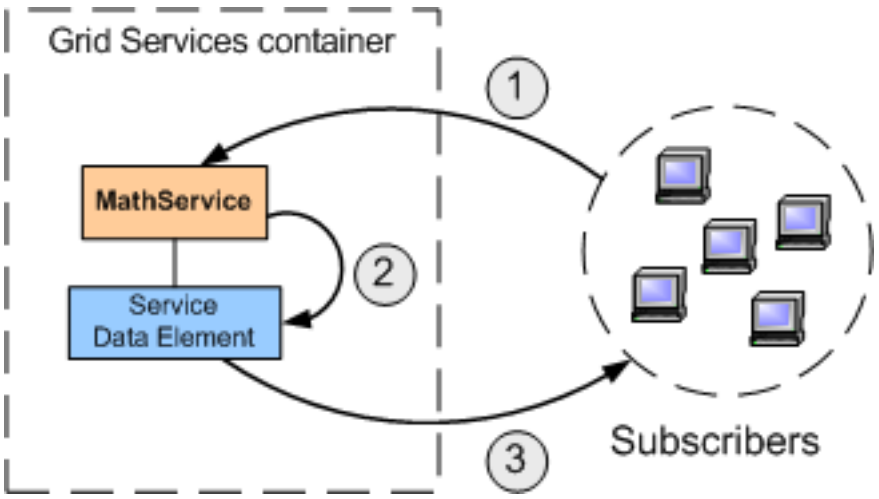
- Each observer needs to get different information from the observable once a change occurs.
- When a notification doesn't imply that the observer will request data from the observable (the observer might choose to ignore the notification)

- **Push**: In this approach, we allow data to travel along with the notification. In our example, the notification would include all the e-mail addresses. This approach is useful when:
  - Each observer needs to obtain the same information once a change occurs.

In general, the pull approach gives the client more control over the data it will get after the notification. The push approach limits what information the client receives, but is more efficient since we save an extra trip to the observable.

Notifications in GT3

Notifications in GT3 are closely related to service data. In fact, the observers don't subscribe to a whole service, but to a particular Service Data Element (SDE) in that service. The following diagram shows how a MathService has a single SDE, and how several clients subscribe to it.



1. **addListener**: This call subscribes the calling client to a particular SDE (which is specified in the call)
2. **notifyChange**: Whenever a change happens, the MathService will ask the SDE to notify its subscribers.
3. **deliverNotification**: The SDE notifies the subscribers that a change had happened.

This notification sent in the third step includes the actual service data, so the subscriber doesn't need to make any more calls to the service. Of course, this is a push notification pattern. GT3 *only* supports push notifications, although it is possible to implement a pull notification by subscribing to a 'dummy SDE' with no data (consequently, no data would be sent with the notification).

Finally, take into account that (in GT3 jargon) the observables are usually called the *notification sources* and the observers are usually called the *notification sinks*.

A notification service

In this example we'll take the code from the Service Data chapter and add notifications to it. We'll see that, once we have service data defined in the service, adding notifications is very simple.

Defining the service interface

Adding support for notifications *does* affect the interface, since we need our service to expose some notification-related methods to the outer world (for example, we need to provide a `subscribe` method so that clients can subscribe themselves to SDEs). However, we don't need to define any operations ourselves. We just need to extend from a standard portType called NotificationSource which includes all these notification-related operations. If we take the GWSDL from the previous example, we just have to add the following:

```
<gwsdl:portType name="MathPortType" extends="ogsi:GridService ogsi:NotificationSou

  <!-- <operation>s -->

  <!-- <serviceData> -->

</gwsdl:portType>
```

Note

The complete GWSDL file is located in `$TUTORIAL_DIR/schema/progtutorial/MathService_sd_notif/Math.gwsdl`

The only other change is that this new GWSDL file has a new target namespace:

```
<definitions name="MathService" targetNamespace="http://www.globus.org/namespaces/
  xmlns:tns="http://www.globus.org/namespaces/2004/02/progtutorial/MathService_sd_
  xmlns:data="http://www.globus.org/namespaces/2004/02/progtutorial/MathService_sd
  xmlns:ogsi="http://www.gridforum.org/namespaces/2003/03/OGSI"
  xmlns:gwsdl="http://www.gridforum.org/namespaces/2003/03/gridWSDLExtensions"
  xmlns:sd="http://www.gridforum.org/namespaces/2003/03/serviceData"
  xmlns:xsd="http://www.w3.org/2001/XMLSchema"
  xmlns="http://schemas.xmlsoap.org/wsdl/">
```

Since we have a new interface (and, therefore, a new namespace) we need to add the following namespace-to-package mappings to our mappings file:

```
http\://www.globus.org/namespaces/2004/02/progtutorial/MathService_sd_notif=
  org.globus.progtutorial.stubs.MathService_sd_notif

http\://www.globus.org/namespaces/2004/02/progtutorial/MathService_sd_notif/bindin
  org.globus.progtutorial.stubs.MathService_sd_notif.bindings

http\://www.globus.org/namespaces/2004/02/progtutorial/MathService_sd_notif/servic
  org.globus.progtutorial.stubs.MathService_sd_notif.service
```

```
http\://www.globus.org/namespaces/2004/02/progtutorial/MathService_sd_notif/MathSD
  org.globus.progtutorial.stubs.MathService_sd_notif.servicedata
```

Note

These lines can be found in `$TUTORIAL_DIR/namespace2package.mappings`

Service Implementation

The service implementation is practically identical to the previous example's implementation. The only difference is that now we'll tell the MathData SDE to notify all its subscribers each time the add or subtract method is invoked:

```
public void add(int a) throws RemoteException
{
  mathDataValue.setLastOp("Addition");
  incrementOps();
  mathDataValue.setValue(mathDataValue.getValue() + a);
  mathDataSDE.notifyChange();
}
```

Note

The complete implementation can be found in `$TUTORIAL_DIR/org/globus/progtutorial/services/core/notifications/impl/MathImpl.java`

Deployment Descriptor

The deployment descriptor is also very similar to the one used in the previous example, with one important change:

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

  <service name="progtutorial/core/notifications/MathService" provider="Handler" s
    <parameter name="name" value="MathService (with Notifications)"/>
    <parameter name="className" value="org.globus.progtutorial.stubs.MathService_s
    <parameter name="baseClassName"
        value="org.globus.progtutorial.services.core.notifications.impl.MathImpl/>
    <parameter name="schemaPath" value="schema/progtutorial/MathService_sd_notif/M

    <parameter name="operationProviders"
        value="org.globus.ogsa.impl.ogsi.NotificationSourceProvider"/>

    <!-- Start common parameters -->
    <parameter name="allowedMethods" value="*"/>
    <parameter name="persistent" value="true"/>
    <parameter name="handlerClass" value="org.globus.ogsa.handlers.RPCURIProvider"
```

```
  </service>

</deployment>
```

Note

This file is `$TUTORIAL_DIR/org/globus/progtutorial/core/notifications/server-config.wsdd`

When defining the service interface we extended from the NotificationSource portType because we needed some operations that notification sources must provide. But, where do we get the implementation of those operations? Well, this is a good example of how 'implementation by inheritance' (extending from GridServiceImpl) and 'implementation by delegation' (using operation providers) are not mutually exclusive in GT3. We have most of the implementation of our service in the MathImpl class (which extends from GridServiceImpl), and complete the implementation by 'plugging in' an operation provider (NotificationSourceProvider, included with GT3) that *provides* all the notification source functionality.

Compile and deploy

Let's build the service:

```
./tutorial_build.sh \
org/globus/progtutorial/services/core/notifications \
schema/progtutorial/MathService_sd_notif/Math.gwsdl
```

Now, deploy the GAR file and start the service container:

```
ant deploy \
-Dgar.name=$TUTORIAL_DIR/build/lib/org_globus_progtutorial_services_core_notificat

globus-start-container
```

Note

Remember, you have to run this from your GT3 installation directory, with a user with write permissions on that directory.

A notification client

Math Listener

This client is a bit more complex than all the previous clients we've seen. First of all, we can't put all the code in the `main` method. A notification client (or more correctly: a class which is going to receive notifications) must implement a `deliverNotification` method, which is the method that the Grid Service will call when a change is produced.

Also, we're actually going to write two clients. The first one is the important one: it is the one that is going to subscribe to MathService and receive notifications. The second one is a very simple one which calls the `add` method. This way, we can have several 'listener clients' running at the same time, and then see how they are all updated when we run the 'adder client'.

```
package org.globus.progtutorial.clients.MathService_sd_notif;

import org.globus.ogsa.client.managers.NotificationSinkManager;
import org.globus.ogsa.NotificationSinkCallback;
import org.globus.ogsa.impl.core.service.ServicePropertiesImpl;
import org.globus.ogsa.utils.AnyHelper;
import org.gridforum.ogsi.ExtensibilityType;
import org.gridforum.ogsi.HandleType;
import org.gridforum.ogsi.ServiceDataValuesType;

import org.globus.progtutorial.stubs.MathService_sd_notif.servicedata.MathDataType

import java.rmi.RemoteException;

public class ClientListener
    extends ServicePropertiesImpl implements NotificationSinkCallback
{
  public static void main(String[] args)
  {
    try
    {
      // Get command-line arguments
      HandleType GSH = new HandleType(args[0]);
      ClientListener clientListener = new ClientListener(GSH);
    }catch(Exception e)
    {
      System.out.println("ERROR!");
      e.printStackTrace();
    }
  }

  public ClientListener(HandleType GSH) throws Exception
  {
    // Start listening to the MathService
    NotificationSinkManager notifManager = NotificationSinkManager.getManager();
    notifManager.startListening(NotificationSinkManager.MAIN_THREAD);
    String sink  =  notifManager.addListener("MathData", null, GSH, this);
    System.out.println("Listening...");

    // Wait for key press
    System.in.read();


    // Stop listening
    notifManager.removeListener(sink);
    notifManager.stopListening();
    System.out.println("Not listening anymore!");
    }


  public void deliverNotification(ExtensibilityType any) throws RemoteException
  {
    try
    {
      // Service Data has changed. Show new data.
      ServiceDataValuesType serviceData = AnyHelper.getAsServiceDataValues(any);
      MathDataType mathData = (MathDataType) AnyHelper.getAsSingleObject(serviceDa

      // Write service data
      System.out.println("Current value: " + mathData.getValue());
      System.out.println("Previous operation: " + mathData.getLastOp());
      System.out.println("# of operations: " + mathData.getNumOps());
```

```
    }catch(Exception exc)
    {
      System.out.println("ERROR!");
      exc.printStackTrace();
    }
  }
}
```

Note

This file is `$TUTORIAL_DIR/org/globus/progtutorial/clients/MathService_sd_notif/ClientListener.java`

Let's take a close look at the class declaration:

```
public class MathListener
    extends ServicePropertiesImpl implements NotificationSinkCallback
```

Unlike all our previous clients, this one extends from a class and implements an interface.

- `extends ServicePropertiesImpl`: This class is the base class of GridServiceImpl. Why would we want our client to extend from a class which is intended for the server-side of our application? Well, consider this: since our client is going to receive calls from the server...our client is also a server! This might sound a bit confusing, but consider the following rule of thumb in distributed systems: "Clients make calls, servers receive them". According to this definition, our "client" is both a client *and* a server, because it is going to make calls to MathService (to subscribe to the MathData SDE), but also receive them (deliverNotification). So, our client needs a server infrastructure, which the ServicePropertiesImpl class provides.
- `implements NotificationSinkCallback`: This interface must be implemented by classes that wish to subscribe to notifications. One of the requirements of this interface is that we have to implement a deliverNotification method.

The main method is very simple. It receives the only argument to the program (MathService's GSH) and then creates a MathListener class. All the 'listening' is done in the constructor of MathListener. This, of course, is not a good design, but it keeps the code simple enough. More elegant solutions should use threads.

The first interesting snippet of code in the constructor is this:

```
NotificationSinkManager notifManager = NotificationSinkManager.getManager();
notifManager.startListening(NotificationSinkManager.MAIN_THREAD);
String sink  =  notifManager.addListener("MathData", null, GSH, this);
```

The NotificationSinkManager is a class that takes care of the whole subscription process, which is done in two simple steps: telling the manager to "start listening" and then telling it what it has to listen to. The addListener merits special attention. Let's take a closer look at the parameters it receives:

- The Service Data Element we want to subscribe to.
- A timeout (null, in the example; we don't want to stop listening)
- The GSH of the Grid Service that has the Service Data Element we want to subscribe to
- The class which will take care of receiving the notifications (in the example, it is `this`, although we could delegate the notifications to a different class)

Once we're listening, we'll wait for a key press. As soon as you press a key, we'll 'unsubscribe' in two steps: removing the listener we created previously, and then stopping the notification manager listening thread.

```
notifManager.removeListener(sink);
notifManager.stopListening();
```

Now, take a look at deliverNotification. It has an argument called ExtensibilityType any. This is the SDE which is sent along with the notification. As we saw in the Service Data section, we first have to cast the ExtensibilityType into a MathDataType using 'helper' classes.

```
ServiceDataValuesType serviceData = AnyHelper.getAsServiceDataValues(any);
MathDataType mathData = (MathDataType) AnyHelper.getAsSingleObject(serviceData, Ma
```

Now, let's compile and run the client:

```
javac \
-classpath ./build/classes/:$CLASSPATH \
org/globus/progtutorial/clients/MathService_sd_notif/ClientListener.java

java \
-classpath ./build/classes/:$CLASSPATH \
-Dorg.globus.ogsa.schema.root=http://localhost:8080/ \
org.globus.progtutorial.clients.MathService_sd_notif.ClientListener \
http://127.0.0.1:8080/ogsa/services/progtutorial/core/notifications/MathService
```

Notice how we have to define a property called `org.globus.ogsa.schema.root`. This should be set to the base URL of your Grid Services container. If all goes well, you should see a "Listening..." message. Since we're not making any changes to MathService, you're not receiving any notifications yet. Remember, this client is only listening for notifications. Now we have to produce changes using a 'Math Adder' to see how the notifications are delivered. To see how notifications can be delivered to multiple listeners, you can run more than one MathListener at the same time.

Math Adder

The code for the Math Adder is pretty straightforward:

```
package org.globus.progtutorial.clients.MathService_sd_notif;
```

```
import org.globus.progtutorial.stubs.MathService_sd_notif.service.MathServiceGridL
import org.globus.progtutorial.stubs.MathService_sd_notif.MathPortType;
import java.net.URL;

public class ClientAdder
{
  public static void main(String[] args)
  {
    try
    {
      // Get command-line arguments
      URL GSH = new java.net.URL(args[0]);
      int a = Integer.parseInt(args[1]);

      // Get a reference to the Grid Service instance
      MathServiceGridLocator mathServiceLocator = new MathServiceGridLocator();
      MathPortType math = mathServiceLocator.getMathServicePort(GSH);

      // Call remote method 'add'
      math.add(a);
      System.out.println("Added " + a);
    }catch(Exception e)
    {
      System.out.println("ERROR!");
      e.printStackTrace();
    }
    }
}
```

Note

This file is `$TUTORIAL_DIR/org/globus/progtutorial/clients/MathService_sd_notif/ClientAdder.java`

Compile and run it:

```
javac \
-classpath ./build/classes/:$CLASSPATH \
org/globus/progtutorial/clients/MathService_sd_notif/ClientAdder.java

java \
-classpath ./build/classes/:$CLASSPATH \
org.globus.progtutorial.clients.MathService_sd_notif.ClientAdder \
http://127.0.0.1:8080/ogsa/services/progtutorial/core/notifications/MathService \
5
```

If all goes well, you'll see how the listener/s receive/s a notification each time you run MathAdder. The internal value, along with then 'number of operations', should increase with each run of MathAdder. Since we're only adding, the 'previous operation' will always be 'Addition'.

Chapter 7. Transient Services

Up to this point, we've seen some of the most exciting features of grid services: statefulness, service data, notifications, portType extension,... However, we're still missing one the most important features included in the OGSI standard: the possibility of creating *transient* services by using a factory/instance pattern.

This might be a good time to reread What is a Grid Service?, where this particular feature was described in great detail.

Adding transience to MathService

The Deployment Descriptor

Turning a service into a factory of transient services is one the easiest things to do in GT3. What we're going to do is take the example from the first chapter, and add transience to it. To do this, we won't need to modify the service interface and not even the service implementation. The only thing we have to do is modify the deployment descriptor.

In fact, if you've been following the tutorial from the beginning, the deployment descriptor for the Math-Service factory was already included in the first chapter's WSDD file:

```
$TUTORIAL_DIR/org/globus/progtutorial/services/core/first/server-deploy.wsdd
```

So, if you've already compiled and deployed that example, you inadvertently also deployed the Math-Service factory service (WSDD allows you to deploy more than one service at once). If you haven't, don't worry, the deployment instructions are repeated below for your convenience.

To emphasize the changes that have to be done to turn any service into a factory of transient services, let's review the deployment descriptor from the first chapter:

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

  <service
    name="progtutorial/core/first/MathService" provider="Handler" style="wrapped">
    <parameter name="name" value="MathService"/>

    <parameter name="baseClassName" value="org.globus.progtutorial.services.core.f
    <parameter name="className" value="org.globus.progtutorial.stubs.MathService.M
    <parameter name="schemaPath" value="schema/progtutorial/MathService/Math_servi

    <!-- Start common parameters -->
    <parameter name="allowedMethods" value="*"/>
    <parameter name="persistent" value="true"/>
    <parameter name="handlerClass" value="org.globus.ogsa.handlers.RPCURIProvider"
  </service>

</deployment>
```

Note

This is a part of as `$TUTORIAL_DIR/org/globus/progtutorial/services/core/first/server-deploy.wsdd`

The following is the deployment descriptor for the exact same service using a factory/instance model. Remember: this means that we'll have a *persistent* factory service which we'll use to create *transient* MathService instances (i.e. MathServices which we'll be able to create and destroy any time we want)

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

  <service name="progtutorial/core/first/MathFactoryService" provider="Handler" st
    <parameter name="name" value="MathService Factory"/>
    <parameter name="instance-name" value="MathService Instance"/>
    <parameter name="instance-schemaPath" value="schema/progtutorial/MathService/M
    <parameter name="instance-baseClassName" value="org.globus.progtutorial.servic
    <parameter name="instance-className" value="org.globus.progtutorial.stubs.Math

    <!-- Start common parameters -->
    <parameter name="allowedMethods" value="*"/>
    <parameter name="persistent" value="true"/>
    <parameter name="handlerClass" value="org.globus.ogsa.handlers.RPCURIProvider"

    <parameter name="className" value="org.gridforum.ogsi.Factory"/>
    <parameter name="baseClassName" value="org.globus.ogsa.impl.ogsi.GridServiceIm
    <parameter name="schemaPath" value="schema/ogsi/ogsi_factory_service.wsdl"/>
    <parameter name="operationProviders" value="org.globus.ogsa.impl.ogsi.FactoryP
    <parameter name="factoryCallback" value="org.globus.ogsa.impl.ogsi.DynamicFact
  </service>

</deployment>
```

Note

This is a part of as `$TUTORIAL_DIR/org/globus/progtutorial/services/core/first/server-deploy.wsdd`

Let's compare both deployment descriptors. First of all, notice how we've added an "instance-" prefix to the original name, schemaPath, baseClassName, and className parameters. By doing this, we're telling the container that these will be the parameters for the instance services (the transient MathServices we'll create through the factory).

Then, we've added a bunch of parameters in the 'common parameters block'. Notice how schemaPath, baseClassName, and className now refer to factory code (provided by GT3), and not to any code we've programmed ourselves. All the code needed to implement the factory/instance model is already included with GT3, so we don't have to worry about programming it ourselves.

So, adding transience to any of the service's we've written is as simple as adding that new 'common parameters block' and putting the "instance-" prefix on any parameter that refers to our own code.

Finally, remember that if you've already deployed the first example in the tutorial, the factory example was also deployed then, so there's no need to rebuild and redeploy it. However, if you haven't built and

deployed the first service, here's the necessary commands in a nutshell:

```
./tutorial_build.sh \
org/globus/progtutorial/services/core/first \
schema/progtutorial/MathService/Math.gwsdl


ant deploy \
-Dgar.name=$TUTORIAL_DIR/build/lib/org_globus_progtutorial_services_core_first.gar
```

A simple client

Since the service interface is exactly the same as the one used in the first chapter, we can reuse the MathService client:

```
$TUTORIAL_DIR/org/globus/progtutorial/clients/MathService/Client.java
```

However, this client only works with services that have the MathService interface. In this example, the services that will have the MathService interface will be the transient instances, so we need to create an instance first through the factory service. We'll see how to do this from Java in the next client. Right now, we'll simply use a GT3 command-line client:

```
ogsi-create-service \
http://127.0.0.1:8080/ogsa/services/progtutorial/core/first/MathFactoryService
```

Note

As with globus-start-container, you need to setup the GT3 command-line clients for this command to work. If you haven't done so yet, take a look at the following section in the How to... appendix: How to... appendix: How to setup the GT3 scripts.

You should see the following:

```
Service successfully created:
Handle: http://127.0.0.1:8080/ogsa/services/progtutorial/core/first/MathFactorySer
Termination Time: infinity
```

The GSH that ogsi-create-service returns is the instance's handle. *This* is the service that has the MathService interface and which we'll be able to invoke using the MathService client. Notice how the GSH ends with a hash number. This hash number will be different each time you run ogsi-create-service, so be sure to use the GSH returned by that command (i.e. don't copy & paste the GSH as it appears here in the tutorial).

That said, let's try out the MathService client with the instance.

```
java \
-classpath ./build/classes/:$CLASSPATH \
org.globus.progtutorial.clients.MathService.Client \
http://127.0.0.1:8080/ogsa/services/progtutorial/core/first/MathFactoryService/has
5
```

Just like in the first example, you should see the following:

```
Added 5
Current value: 5
```

If you create several instances, you'll be able to see that each instance keeps its own particular state. In other words, what we do to one instance affects that instance (and that instance alone). This is why using a factory/instance model is interesting when a client wants to dynamically create a service, use it for a particular task, and destroy the instance once it's done with its work.

You can destroy the instance by using the ogsi-destroy-instance command:

```
ogsi-destroy-service \
http://127.0.0.1:8080/ogsa/services/progtutorial/core/first/MathFactoryService/has
```

A slight less simple client

This second client doesn't connect to an existing instance. Instead, it first connects to the MathService factory, requests that a new instance be created, uses it, and then destroys it. This example has some important differences with respect to prior examples, so besides just printing the whole source code, we'll see some important lines in more detail.

```
package org.globus.progtutorial.clients.MathService;

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

import org.globus.progtutorial.stubs.MathService.service.MathServiceGridLocator;
import org.globus.progtutorial.stubs.MathService.MathPortType;

import java.net.URL;

public class FactoryClient
{
  public static void main(String[] args)
  {
    try
    {
      // Get command-line arguments
      URL GSH = new java.net.URL(args[0]);
      int a = Integer.parseInt(args[1]);

      // Get a reference to the MathService Factory
      OGSIServiceGridLocator gridLocator = new OGSIServiceGridLocator();
      Factory factory = gridLocator.getFactoryPort(GSH);
      GridServiceFactory mathFactory = new GridServiceFactory(factory);

      // Create a new MathService instance and get a reference
      // to its Math PortType
      LocatorType locator = mathFactory.createService();
      MathServiceGridLocator mathLocator =   new MathServiceGridLocator();
```

```
      MathPortType math = mathLocator.getMathServicePort(locator);

      // Call remote method 'add'
      math.add(a);
      System.out.println("Added " + a);

      // Get current value through remote method 'getValue'
      int value = math.getValue();
      System.out.println("Current value: " + value);

      // Destroy the instance
      math.destroy();
    }catch(Exception e)
    {
      System.out.println("ERROR!");
      e.printStackTrace();
    }
  }
}
```

Note

This file is `$TUTORIAL_DIR/org/globus/progtutorial/clients/MathService/FactoryClient.java`

First off, let's take a look at how we make the connection to the MathService Factory:

```
OGSIServiceGridLocator gridLocator = new OGSIServiceGridLocator();
Factory factory = gridLocator.getFactoryPort(GSH);
GridServiceFactory mathFactory = new GridServiceFactory(factory);
```

This should look slightly familiar. Remember how, in previous examples, we could connect to MathService with just two lines (which look suspiciously like these two... just replace OGSIServiceGridLocator with MathServiceGridLocator and Factory with MathPortType). Remember: factories are themselves Grid Services. What the previous three lines do is obtain a reference to a factory. The following lines show how we can create a new MathService instance with that factory:

```
LocatorType locator = mathFactory.createService();
MathServiceGridLocator mathLocator =   new MathServiceGridLocator();
MathPortType math = mathLocator.getMathService(locator);
```

First of all we call the createService method, which is a factory operation. This method returns the instance's *locator*, which we will use to get a reference to the Math PortType in the remaining two lines. After these three lines, we are ready to call the instance. These calls are exactly the same as the previous client. However, after we've made the calls, we have to destroy the instance. This is done by making a call to the destroy() method in the GridService portType. This method orders a Grid Service to destroy itself. Since MathPortType extends from GridService (as all Grid Services must extend), we can invoke the destroy method directly on our MathPortType.

```
math.destroy();
```

Let's give this new client a try:

```
javac \
-classpath ./build/classes/:$CLASSPATH \
org/globus/progtutorial/clients/MathService/FactoryClient.java
```

Now, let's run it. Remember, we don't have to create an instance because the client takes care of it. Also, the URL we pass as an argument is the factory GSH, *not* an instance GSH.

```
java \
-classpath ./build/classes/:$CLASSPATH \
org.globus.progtutorial.clients.MathService.FactoryClient \
http://127.0.0.1:8080/ogsa/services/progtutorial/core/first/MathFactoryService \
5
```

If you run this program several times, you'll notice that the internal value is always the same. This is because each time we run the client, we're using a different instance (so each time we run the client, the instance always has an initial value of 0). Of course, in this example this might not make much sense. However, imagine that MathService offers dozens of complicated operations which require intermediate values to be remembered and that, once you've arrived at a final result, you have no more need for the instance. In that case, it makes sense to destroy the instance as soon as we're done with it, so we can free system resources.

Chapter 8. Logging

One of the interesting features in the Globus Toolkit 3 is the possibility of writing a log of interesting events (warnings, errors, debug information, etc.) to the console or to a file. This feature is based on the Apache Jakarta Commons Logging [http://jakarta.apache.org/commons/logging/] component. In this section we'll add logging to the very first example of the tutorial. You can, if you wish, follow this example by modifying the first example, recompiling it, and redeploying it. However, all the instructions will be given relative to a new directory where the modified first example (with logging) can be found:

```
$TUTORIAL_DIR/org/globus/progtutorial/services/core/logging/
```

The Jakarta Commons Logging architecture

The goal of the Apache Jakarta Commons Project [http://jakarta.apache.org/commons/] is the development of reusable Java components, such as validation classes, command line option parsers, etc. Once of the components in this project is the Commons Logging [http://jakarta.apache.org/commons/logging/] component, which allows us to easily produce a log from our Java class.

The Commons Logging component has 6 levels of logging. This means that we are not limited to just one type of log message, but 6 types of messages with varying degree of 'severity'. This allows us to filter the types of messages, so we have a log with only the information we want. For example, at one point we might be interested in producing a log with all the debugging information, but later on we will probably only want a log with errors and warnings produced by our program.

The six levels of log messages are:

- **Debug**
- **Trace**
- **Info**
- **Warn**
- **Error**
- **Fatal**

What messages go into each category is entirely up to the programmer, as we'll see in the next page.

Adding logging to MathService

Enabling logging in our Grid Service is very easy. The following code shows all the necessary changes in bold. We'll take a closer look in a moment.

```
// ...

import org.apache.commons.logging.Log;
import org.apache.commons.logging.LogFactory;
```

```
// ...

public class MathImpl extends GridServiceImpl implements MathPortType
{

  // Create this class's logger
  static Log logger = LogFactory.getLog(MathImpl.class.getName());

  // ...

  public void add(int a) throws RemoteException
  {

    logger.info("Addition invoked with parameter a=" + String.valueOf(a));
    if (a==0)
      logger.warn("Adding zero doesn't modify the internal value!");
    value = value + a;
  }

  public void subtract(int a) throws RemoteException
  {

    logger.info("Subtraction invoked with parameter a=" + String.valueOf(a));
    if (a==0)
      logger.warn("Subtracting zero doesn't modify the internal value!");
    value = value - a;
  }

  public int getValue() throws RemoteException
  {
    logger.info("getValue() invoked");
    return value;
  }
}
```

Note

Add these modifications to `$TUTORIAL_DIR/org/globus/progtutorial/services/core/first/impl/MathImpl.java` and save it as `$TUTORIAL_DIR/org/globus/progtutorial/services/core/logging/impl/MathImpl.java`

As you can see, the modifications are very simple. First of all, we need to import two packages from the Commons Logging component. After that, we need to create a static Log attribute. This attribute is created using a LogFactory. Notice how we have to pass the name of our class to the getLog method.

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

After these two modifications, our MathImpl class is ready to do some serious logging. We're going to generate some Info and Warn messages. This is as simple as calling the info or warn method in the logger static attribute. The only necessary parameter is the message we want to write in the log.

```
logger.info("Addition invoked with parameter a=" + String.valueOf(a));
```

```
if (a==0)
  logger.warn("Adding zero doesn't modify the internal value!");
```

As you can see, we're generating an info message every time any of the methods is invoked, and a warning message whenever the add or subtract method is called with an argument equal to zero.

Of course, you can generate messages in any of the other levels by calling:

- logger.debug("Message")
- logger.trace("Message")
- logger.error("Message")
- logger.fatal("Message")

Writing the deployment descriptor

The deployment descriptor will be very similar to the first example's descriptor. The only important change is the GSH of the service and the baseClassName of the service (which is the class we've just programmed: the first example's class plus logging code)

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

  <service name="progtutorial/core/logging/MathService" provider="Handler" style="
    <parameter name="name" value="MathService"/>
    <parameter name="className" value="org.globus.progtutorial.stubs.MathService.M
    <parameter name="baseClassName" value="org.globus.progtutorial.services.core.l
    <parameter name="schemaPath" value="schema/progtutorial/MathService/Math_servi

    <!-- Start common parameters -->
    <parameter name="allowedMethods" value="*"/>
    <parameter name="persistent" value="true"/>
    <parameter name="handlerClass" value="org.globus.ogsa.handlers.RPCURIProvider"
  </service>

</deployment>
```

Note

This file is `$TUTORIAL_DIR/org/globus/progtutorial/services/core/logging/server-deploy.wsdd`

Generate GAR and deploy

Let's compile the service:

```
./tutorial_build.sh \
org/globus/progtutorial/services/core/logging \
schema/progtutorial/MathService/Math.gwsdl
```

And deploy it:

```
ant deploy \
-Dgar.name=$TUTORIAL_DIR/build/lib/org_globus_progtutorial_services_core_logging.g
```

Viewing log output

Our MathImpl class is now ready to generate logs. But, where do all those log messages end up? Well, we can either write them to the console output of the Grid Services container, or write them in a file. This is specified in a file you'll find at the root of your GT3 installation: `$GLOBUS_LOCATION/ogsilogging.properties`. Just add the following line at the end of that file:

```
org.globus.progtutorial.services.core.logging.impl.MathImpl=console,info
```

Each line of that file specifies how logging must be handled in those classes that are logging-enabled. In our case, the class where we've enabled logging is `org.globus.progtutorial.services.core.providers.impl.MathImpl`. The two options after the equals sign (=) tell the logger where it should write the messages, and how to filter them (according to their level).

The messages can either be written to the console, by specifying the console option, or to a file, by writing the file name. This file will be created in the directory specified in another file called `$GLOBUS_LOCATION/ogsilogging_parms.properties` (in the option logDestinationBasePath).

The filtering option can take any of the following values:

- **all** or **debug**
- **trace**
- **info**
- **warn**
- **error**
- **fatal**
- **off**

Each of these correspond to the logging levels we saw earlier. We can also ask for all or none of the messages to be displayed. Take into account that you can only specify *one* level of filtering. For example, if you select the warn level, you will get all the messages generated at that level and at 'more severe' levels (error and fatal). The logic behind this is that usually you don't want the log message from one specific level, but all the messages which have *at least* a certain severity (if you're interested in the warnings, you're probably also interested in the errors and the fatal exceptions).

Let's put all this to the test. Since we haven't changed the interface of the service we can, once again, reuse the MathService client we've used in previous examples.

```
java \
-classpath ./build/classes/:$CLASSPATH \
org.globus.progtutorial.clients.MathService.Client \
http://127.0.0.1:8080/ogsa/services/progtutorial/core/logging/MathService \
5
```

Since we've set the log level to info, we should get both Info and Warn messages. You should see this in the console output of the services container:

```
[DATE TIME ] CLASS_NAME [add:39] INFO: Addition invoked with parameter a=5
[DATE TIME ] CLASS_NAME [getValue:55] INFO: getValue() invoked
```

Each log entry includes the date and time of the entry, plus the name of the class which produced the log entry (in our case org.globus.progtutorial.services.core.providers.impl.MathImpl) We didn't get any Warn message because we need to invoke the add method with the value zero. Let's try to do that:

```
java \
-classpath ./build/classes/:$CLASSPATH \
org.globus.progtutorial.clients.MathService.Client \
http://127.0.0.1:8080/ogsa/services/progtutorial/core/logging/MathService \
0
```

You should see this in the console:

```
[DATE TIME ] CLASS_NAME [add:39] INFO: Addition invoked with parameter a=0
[DATE TIME ] CLASS_NAME [add:41] WARN: Adding zero doesn't modify the internal val
[DATE TIME ] CLASS_NAME [getValue:55] INFO: getValue() invoked
```

Finally, let's try changing the log level in the `$GLOBUS_LOCATION/ogsilogging.properties file`:

```
org.globus.progtutorial.services.core.logging.impl.MathImpl=console,
warn
```

You will need to restart the container for this change to have effect. Once you've done so, try running the client again (passing a zero as the value to add). You should see this in the console:

```
[DATE TIME ] CLASS_NAME [add:41] WARN: Adding zero doesn't modify the internal val
```

Since the log level is Warn, this means that the logger will only output messages which are 'at least as severe as a warning'. Since an Info message is not as severe as a Warn message, it will not pass the fil-

ter.

Chapter 9. Lifecycle Management

Since OGSI includes a factory/instance model, we are faced with services which have *non-trivial* lifecycles. In plain web services, the lifecycle of a service was pretty simple: the service is created when the container is started, and it is destroyed when the container is stopped. Managing its lifecycle wasn't really a big deal. However, since Grid Services are *potentially transient*, and we can have instances popping in and out of our container at any given time, the management and control of their lifecycle is no longer trivial.

But, what exactly do we refer to when we say 'lifecycle'? In most object systems, instances are usually said to have a *lifecycle*. On one hand, this refers to the time between instance creation and destruction. On the other hand, *lifecycle* is also understood in a broader sense: some instances will need to outlive not only the lifetime of their clients, but also the lifetime of the server they are contained in. This means that if the server is restarted, the instance should be loaded with the exact same internal values it had right before the server was stopped.

GT3 provides us the necessary tools to manage the lifecycle of Grid Services. For example, we can tell our instance to run some code right before it is created and right before it is destroyed (to load and unload its internal values to secondary storage). In this section we will see some of the lifecycle management tools we can find in GT3.

Since lifecycle is specially important in transient services, we'll add lifecycle management to the transient services example. However, the code we'll use will be a combination of the transient services example and the logging example seen in the previous section. We'll reuse the logging example's implementation (which already includes some logging instructions which will come in handy) but will use a deployment descriptor for transient services. Although much of the code will be presented as a modification of previously existing code, all instructions will be given relative to a new directory that includes all the modified code (with lifecycle management):

```
$TUTORIAL_DIR/org/globus/progtutorial/services/core/lifecycle/
```

The callback methods

One of the ways to manage an instance's lifecycle in GT3 is through *callback methods*. These methods are called at specific points during an instance lifetime (such as instance creation and destruction).

Using callback methods is very simple. Our class must implement the GridServiceCallback interface, which includes all the callback methods. The following class implements all the callback methods:

```
// ...

import org.globus.ogsa.GridServiceCallback;

// ...

public class MathImpl extends GridServiceImpl
                      implements MathPortType, GridServiceCallback
{

  // ...


  // Callback methods
  public void preCreate(GridServiceBase base) throws GridServiceException
```

```
  {
    super.preCreate(base);
    logger.info("Instance is going to be created (preCreate)");
  }

  public void postCreate(GridContext context) throws GridServiceException
  {
    super.postCreate(context);
    logger.info("Instance has been created (postCreate)");
  }

  public void activate(GridContext context) throws GridServiceException
  {
    super.activate(context);
    logger.info("Instance has been activated (activate)");
  }

  public void deactivate(GridContext context) throws GridServiceException
  {
    super.deactivate(context);
    logger.info("Instance has been deactivated (deactivate)");
  }

  public void preDestroy(GridContext context) throws GridServiceException
  {
    super.preDestroy(context);
    logger.info("Instance is going to be destroyed (preDestroy)");
  }
}
```

Note

Add these modifications to `$TUTORIAL_DIR/org/globus/progtutorial/services/core/logging/impl/MathImpl.java` and save it as `$TUTORIAL_DIR/org/globus/progtutorial/services/core/lifecycle/impl/MathImpl.java`

First of all, notice how we need to call the base class callback methods from our own callback methods. For example, in `postCreate`:

```
public void postCreate(GridContext context) throws GridServiceException
{
  super.postCreate(context);
  logger.info("Instance has been created (postCreate)");
}
```

It is important to do this so that the callback methods in `GridServiceImpl` (the base class) also get called. The `GridServiceImpl` callback methods take care of initializing a lot of internal values (including service data), so failing to call them will probably produce unexpected results.

Note

However, we don't need to call the base class callback methods when using operation providers. Remember that operation providers don't extend from `GridServiceImpl`, but that

the `GridServiceImpl` is still present 'in the background' (we specified this in the WSDD file). The grid services container makes sure that both the `GridServiceImpl` and operation provider's callback methods are called when necessary.

Now, let's take a closer look at when each callback method is called:

- **preCreate** : This method is called when a Grid Service starts the creation process. At this stage, its configuration has not been loaded.
- **postCreate** : This method is called when a service has been created and all of its configuration has been set up.
- **activate** : All Grid Services are in a 'deactivated' state by default (which means they still haven't been loaded into memory). This callback method is invoked when a service is activated.
- **deactivate** : This method is called before a service is deactivated.
- **preDestroy** : This method is called right before a service is destroyed.

Writing the deployment descriptor

The deployment descriptor will be very similar to the transient service's descriptor. The only important change is the GSH of the service and the baseClassName of the service (which is the class we've just programmed: the logging example's class plus the callback methods)

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

  <service name="progtutorial/core/lifecycle/MathFactoryService" provider="Handler
    <parameter name="name" value="MathService Factory"/>
    <parameter name="instance-name" value="MathService Instance"/>
    <parameter name="instance-schemaPath" value="schema/progtutorial/MathService/M
    <parameter name="instance-baseClassName" value="org.globus.progtutorial.servic
    <parameter name="instance-className" value="org.globus.progtutorial.stubs.Math

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

</deployment>
```

Note

This file is `$TUTORIAL_DIR/org/globus/progtutorial/services/core/lifecycle/server-deploy.wsdd`

Compiling, deploying, and trying it out

Before compiling this service, since we have a new class that produces logging information, we need to enable logging for that class:

```
org.globus.progtutorial.services.core.lifecycle.impl.MathImpl=console,info
```

Note

Add this line at the end of `$GLOBUS_LOCATION/ogsilogging.properties`

Now, let's build the service:

```
./tutorial_build.sh \
org/globus/progtutorial/services/core/lifecycle \
schema/progtutorial/MathService/Math.gwsdl
```

And deploy it:

```
ant deploy \
-Dgar.name=$TUTORIAL_DIR/build/lib/org_globus_progtutorial_services_core_lifecycle
```

Testing the service

Since our service uses a factory/instance model, the first thing we need to do is create an instance:

```
ogsi-create-service \
 http://127.0.0.1:8080/ogsa/services/progtutorial/core/lifecycle/MathFactoryServic
```

If you look at the server-side logs, you'll see the following: (verbosity removed for comfort)

```
INFO: Instance is going to be created (preCreate)
INFO: Instance has been created (postCreate)
```

Now, let's access the service instance using the MathService client:

```
java \
-classpath ./build/classes/:$CLASSPATH \
org.globus.progtutorial.clients.MathService.Client \
http://127.0.0.1:8080/ogsa/services/progtutorial/core/lifecycle/MathFactoryService
5
```

Remember to use the instance GSH returned by ogsi-create-service You should now see the following in the server-side logs:

```
INFO: Addition invoked with parameter a=5
```

```
INFO: getValue() invoked
```

These are the log messages produced by the `add` and `getValue` methods, which means no callback methods have been invoked while calling add and getValue.

Now, let's destroy the instance:

```
ogsi-destroy-service \
http://127.0.0.1:8080/ogsa/services/progtutorial/core/lifecycle/MathFactoryService
```

Remember to use the instance GSH returned by ogsi-create-service You should now see the following in the server-side logs:

```
INFO: Instance is going to be destroyed (preDestroy)
```

The lifecycle monitor

The callback methods we just saw are pretty nice, but they're not very reusable. Imagine you wanted to use the same preCreate and postCreate methods in several Grid Services. The only way to do this would be to make a base class with those two common methods, and have all your Grid Services extend that class. Of course, this is a very strong restriction, since our Grid Service might already extend from an existing class (as we saw in the Operation Providers section).

The solution to this in GT3 is the *lifecycle monitor*. A lifecycle monitor is a class that implements ServiceLifecycleMonitor, an interface with callback methods that are called at specific points in a Grid Service's lifetime. We won't need to extend from this class, or even reference it directly from our code. We'll just add a line to our deployment descriptor saying that we want a certain lifecycle monitor to be called when those special events happen. Of course, we can use the same lifecycle monitor in different Grid Services (including it in their deployment descriptors).

The following would be a very basic lifecycle monitor class, which simply writes messages to a log:

```
package org.globus.progtutorial.services.core.lifecycle.impl;

import org.globus.ogsa.GridServiceException;
import org.globus.ogsa.ServiceLifecycleMonitor;
import org.globus.ogsa.GridContext;

import org.apache.commons.logging.Log;
import org.apache.commons.logging.LogFactory;

public class MathLifecycleMonitor implements ServiceLifecycleMonitor
{
  // Create this class's logger
  static Log logger = LogFactory.getLog(MathLifecycleMonitor.class.getName());

  public void create(GridContext context) throws GridServiceException
  {
    logger.info("Instance is going to be created (create)");
  }

  public void destroy(GridContext context) throws GridServiceException
  {
```

```
    logger.info("Instance is going to be destroyed (destroy)");
  }

  public void preCall(GridContext context) throws GridServiceException
  {
    logger.info("Service is going to be invoked (preCall)");
  }

  public void postCall(GridContext context) throws GridServiceException
  {
    logger.info("Service invocation has finished  (postCall)");
  }

  public void preSerializationCall(GridContext context)
  {
    logger.info("Input parameters are going to be deserialized (preSerializationCa
  }

  public void postSerializationCall(GridContext context)
  {
    logger.info("Input parameters have been deserialized (postSerializationCall)")
  }
}
```

Note

This file is `$TUTORIAL_DIR/org/globus/progtutorial/services/core/lifecycle/impl/MathLifecycleMonitor.java`

To make sure the log messages are printed out, you need to add the following line to the $GLOBUS_LOCATION/ogsilogging.properties file:

```
org.globus.progtutorial.services.core.lifecycle.impl.MathLifecycleMonitor=console,
```

To tell the Grid Services container that you want it to use this lifecycle monitor, you need to add the following parameter to the deployment descriptor:

```
<parameter name="lifecycleMonitorClass"
  value="org.globus.progtutorial.services.core.lifecycle.impl.MathLifecycleMonitor
```

To try this out, just recompile and redeploy the Grid Service just as we did in the previous page. If you create an instance, access it with the MathService client, and destroy the instance, you should see the following in the server-side logs: (logs produced by the lifecycle monitor are highlighted in bold)

```
INFO: Instance is going to be created (preCreate)
INFO: Instance has been created (postCreate)
INFO: Instance is going to be created (create)

INFO: Input parameters are going to be deserialized (preSerializationCall)
```

```
INFO: Service is going to be invoked (preCall)
INFO: Addition invoked with parameter a=5
INFO: Service invocation has finished  (postCall)
INFO: Input parameters have been deserialized (postSerializationCall)

INFO: Input parameters are going to be deserialized (preSerializationCall)
INFO: Service is going to be invoked (preCall)
INFO: getValue() invoked
INFO: Service invocation has finished  (postCall)
INFO: Input parameters have been deserialized (postSerializationCall)

INFO: Input parameters are going to be deserialized (preSerializationCall)
INFO: Service is going to be invoked (preCall)
INFO: Instance is going to be destroyed (preDestroy)
INFO: Instance is going to be destroyed (destroy)
INFO: Service invocation has finished  (postCall)
INFO: Input parameters have been deserialized (postSerializationCall)
```

Lifecycle parameters in the deployment descriptor

We can control some lifecycle parameters in the deployment descriptor. For example, the following parameter allows us to control when an instance will be deactivated:

```
<parameter name="instance-deactivation" value="10000"/>
```

The time is expressed in milliseconds. After the instance has been idle for 10 seconds, it will be deactivated. Remember instances can be in an activated or deactivated state. Instances are created in a deactivated state (not loaded into memory), and are activated upon their first use. By default, they remain activated indefinitely. In some cases, it might be interesting to unload the instances after a certain time to save system resources. Of course, once the instance is invoked again, it will once again be activated.

Part III. GT3 Security Services

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: the **GT3 Security Services**. This part of the tutorial assumes that the reader knows his way around GT3 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://www.casa-sotomayor.net/gt3-tutorial] to try out the services by yourself.

Table of Contents

Chapter 10. Fundamental Security Concepts

Working with the GT3 Security Services 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 section. 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:

- Books:
  - *Practical Cryptography*. Bruce Schneier. John Wiley & Sons, 2003. Visit web site [http://www.schneier.com/book-practical.html].
  - *Applied Cryptography*. Bruce Schneier. John Wiley & Sons, 1996. Visit web site [http://www.schneier.com/book-applied.html].
- Wikipedia articles:
  - Information Security [http://en.wikipedia.org/wiki/Information_security]
  - Secure Computing [http://en2.wikipedia.org/wiki/Computer_security]
  - Public-key Cryptography [http://en.wikipedia.org/wiki/Public-key_cryptography]
  - Certificate Authority [http://en2.wikipedia.org/wiki/Certificate_authority]

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 the tutorial.

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 page.

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.

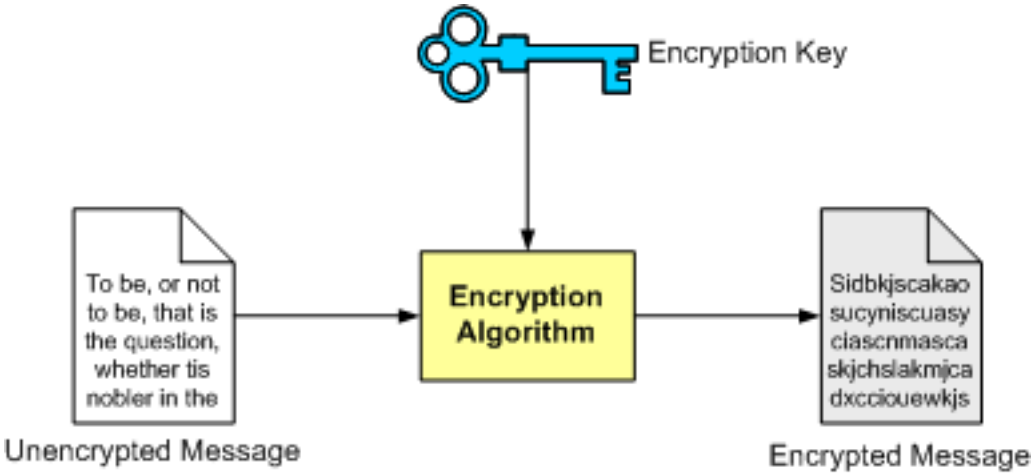
Introduction to cryptography

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.

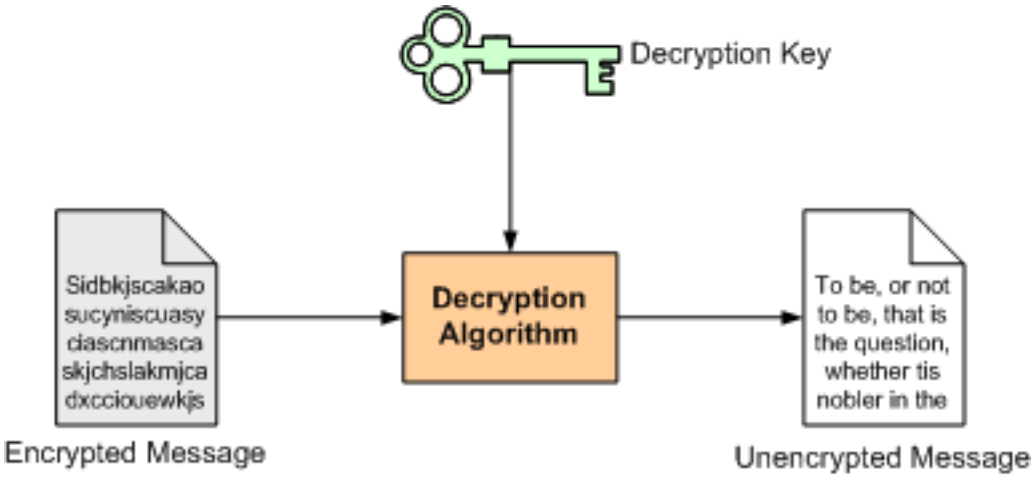
Key-based algorithms

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 by the *previous* letter in the alphabet. These kind of algorithms, based on the *substitution* of letters, are easily cracked. Most modern algorithms 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':



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':



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 me 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:

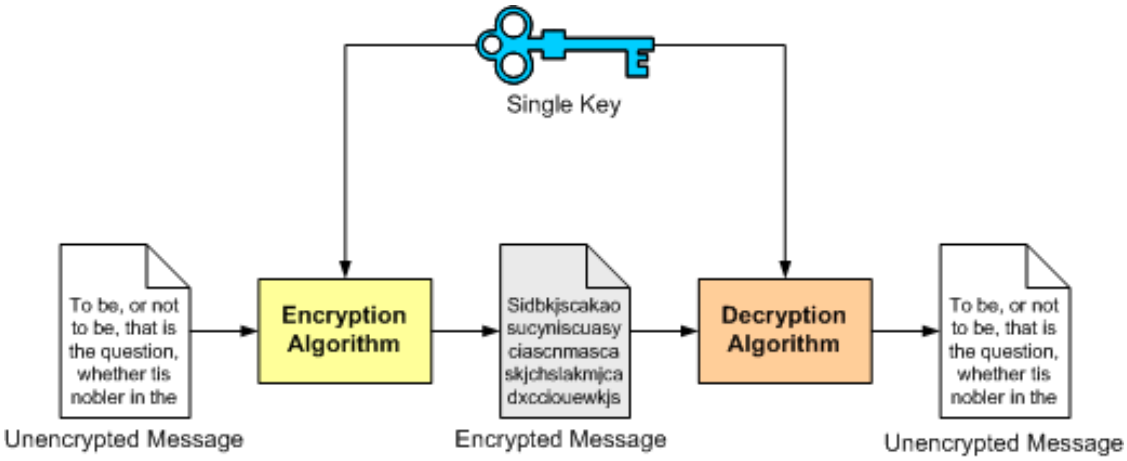
```
  5 4 6 6 9 8 8 6 7 4 4
- 4 2 3 2 4 2 3 2 4 2 3
  ---------------------
  1 2 3 4 5 6 5 4 3 2 1
```

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 at least 128 bits long, 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:



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 page, are the most commonly used type of asymmetric algorithms.

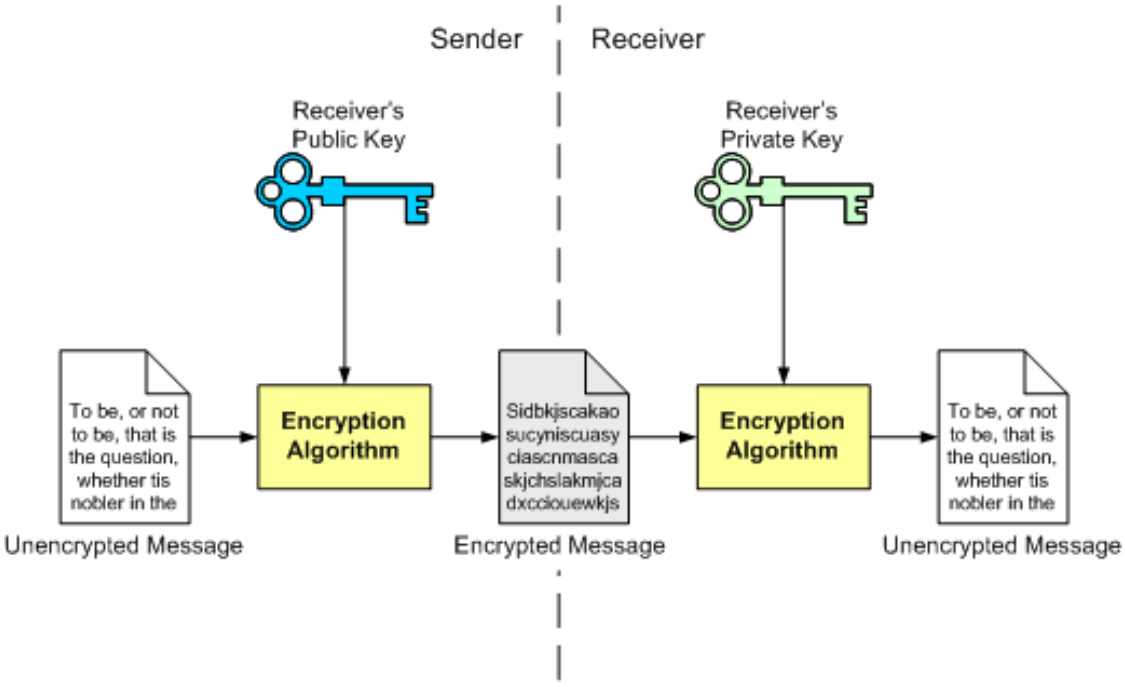
Public key cryptography

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.

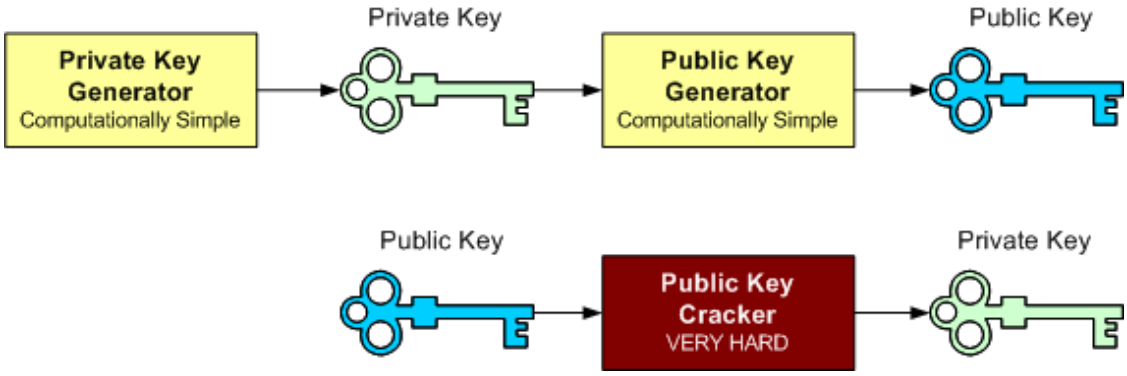
A secure conversation using public-key cryptography

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.



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.

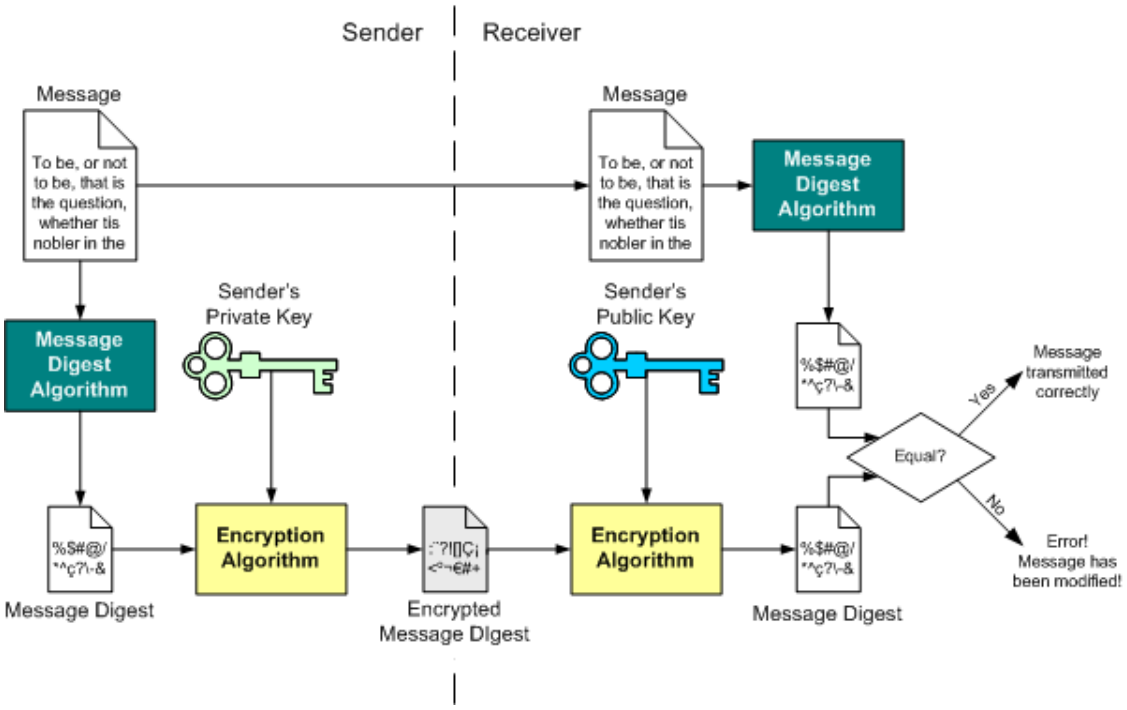


Another important advantage is that, unlike symmetric algorithms, public-key systems can guarantee integrity and authentication, not only privacy. The basic communication seen above only guarantees privacy. We will shortly see how integrity and authentication fit into public-key systems.

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)



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.

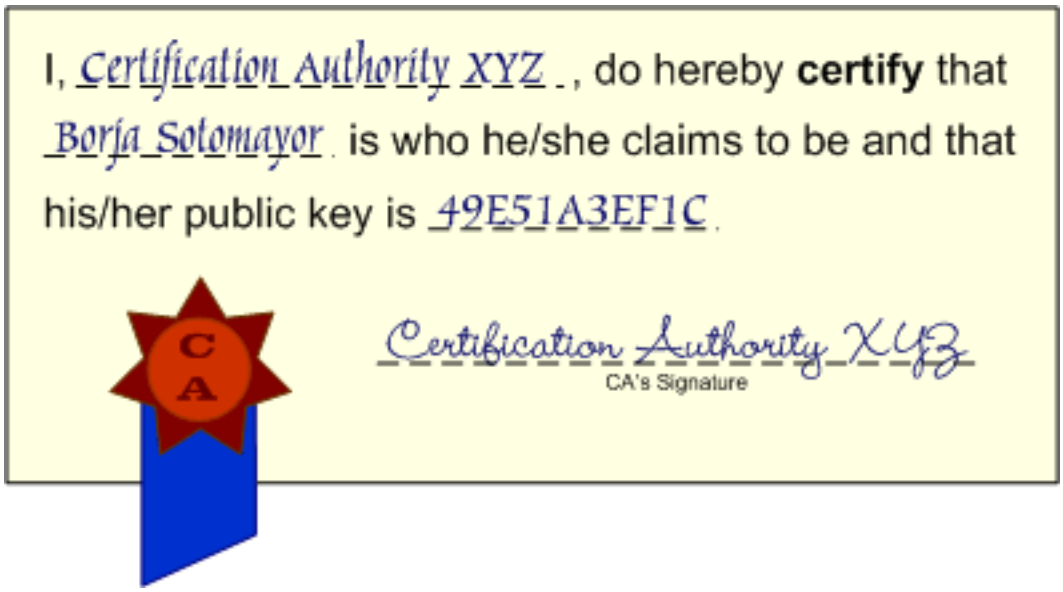
Authentication in public-key systems

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). The following illustration might help get an idea of what a digital certificate is:



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, this 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).

Distinguished names

Names in X.509 certificates are not encoded simply as 'common names', such as "Borja Sotomayor", or "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 my distinguished name:

```
O=University of Deusto, OU=Department of Software Engineering, CN=Borja Sotomayor
```

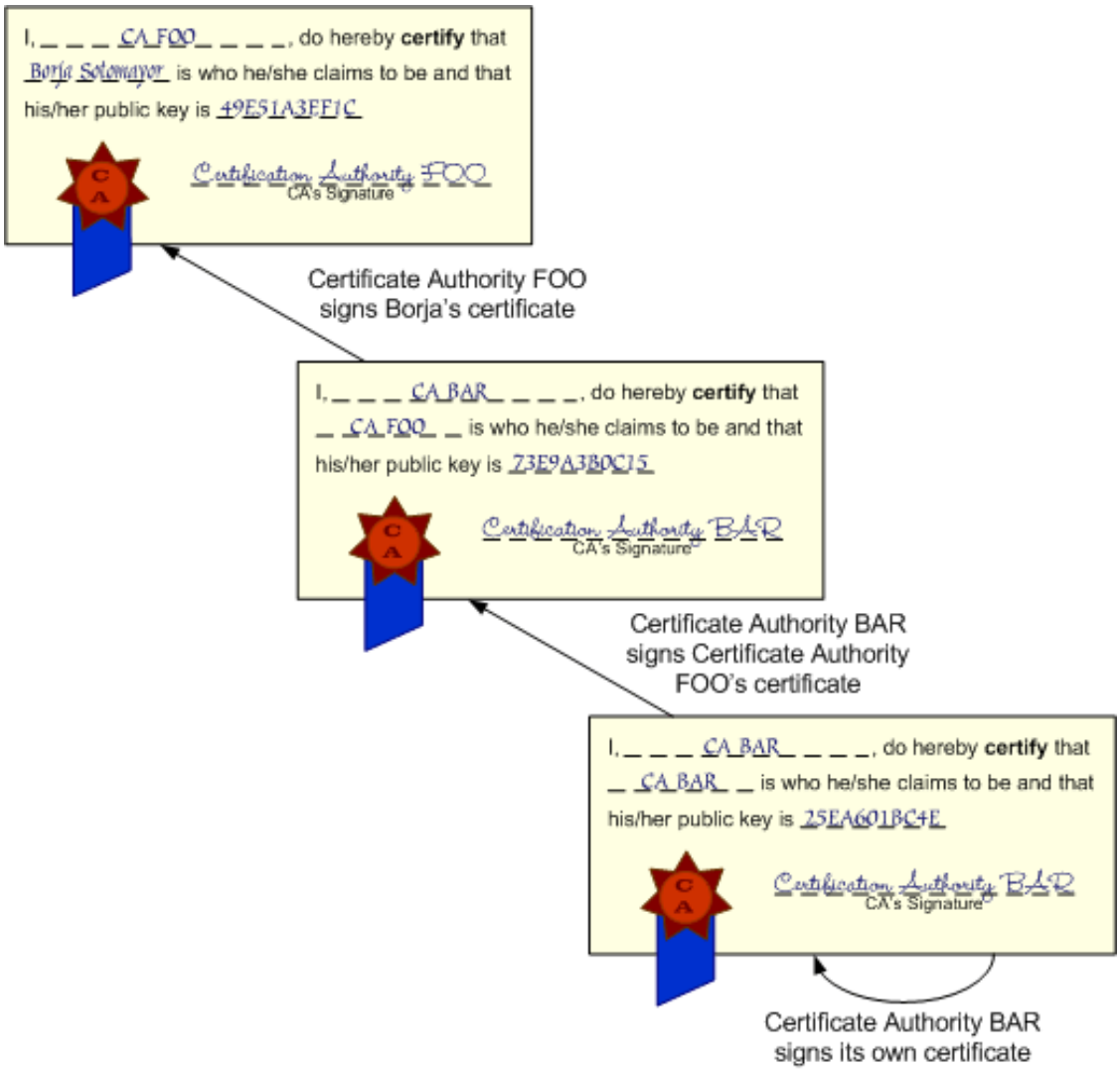
So what do "O", "OU", and "CN" mean? A distinguished name can have several different attributes, and the most common are the following:

- **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)

The following illustration might make things a bit clearer:



In the illustration, my 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 my certificate, and don't explicitly trust CA FOO (the issuer of my certificate), this doesn't automatically mean my 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 my 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 11. GSI: Grid Security Infrastructure

This chapter introduces the Grid Security Infrastructure, the basis for GT3's Security layer. A working knowledge of fundamental security concepts is assumed in this section. If you've read the previous section (Fundamental Security Concepts), 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. If neither of these apply to you, then I strongly suggest you take a look at the Fundamental Security Concepts chapter.

Introduction to GSI

If you're familiar with Grid Computing (which you should be, if you've come this far in the tutorial! :-) 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 my grid application is properly *authenticated*.
- We're going to bump into some pretty interesting scenarios. For example, suppose organization A asks B to perform a certain task. B, on the other hand, realizes that the task should be delegated to organization C. However, let's suppose C only trusts A (and not B). Should C turn down the request because it comes from B, or accept it since the 'original' requestor is A?
- Depending on my application, I 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 3 allows us to overcome the security challenges posed by grid applications through the *Grid Security Infrastructure* (or GSI), which offers programmers the following three features:

- Complete public-key system
- Mutual authentication through digital certificates
- Credential delegation and single sign-on

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 grid services. It is based on standard technologies, such as TLS (formerly SSL) and secure Web Services specifications (XML-Signature, XML-Encryption, etc.)

Let's take a closer look at the three main features of GSI:

Complete public-key system

The GSI is based on public-key cryptography, and therefore can be configured to guarantee privacy, integrity, and authentication (strong authentication is provided in conjunction with certificates, as will be explained next). 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 grid 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.

Mutual authentication through digital certificates

The GSI uses X.509 certificates (as seen in the previous chapter) to guarantee a strong authentication. *Mutual* authentication simply means that in GSI, both parts of a secure conversation must be authenticated. In other words, when A wants to communicate with B, A must trust B and B must trust A. Remember that 'trust' (in this context) means that A must have the certificate of the CA that signed B's certificate, and vice versa. Otherwise, A won't trust B (and vice versa).

One of the first practical things we will do in the tutorial (in the next section, "Setting up GSI") is to setup a very simple CA, and get a digital certificate for ourselves.

Credential delegation and single sign-on

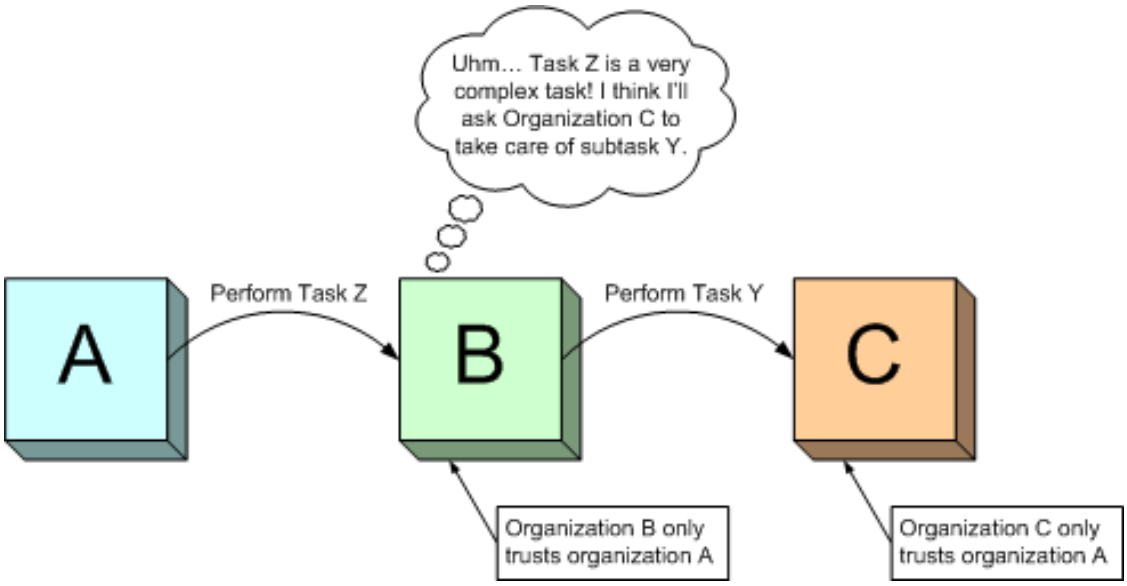
This feature is explained in detail in the next page.

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

In the previous page, an interesting scenario was described:

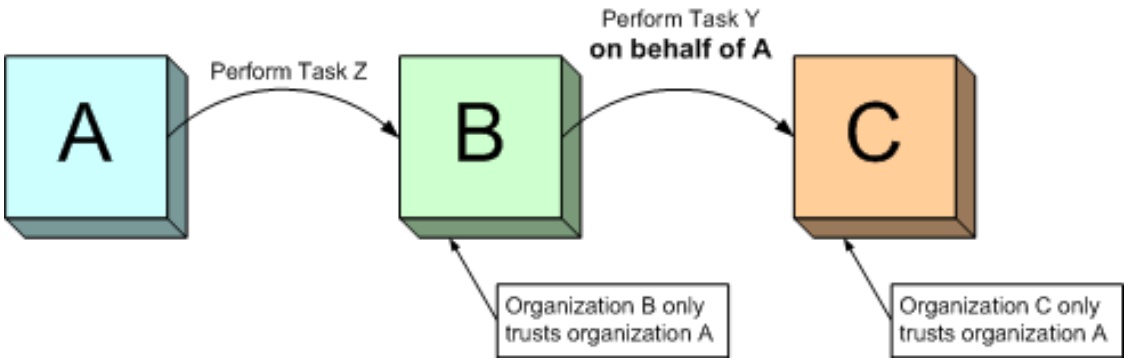


Organization A asks Organization B to perform a task. Since B trusts A, it accepts to perform the task.

But let's suppose that task Z is very complex, and that one of its subtasks (Y) must be performed by a third organization: Organization C. In this case, B will ask C to perform subtask Y but, alas!, C only trusts A. What should C do? It has two options:

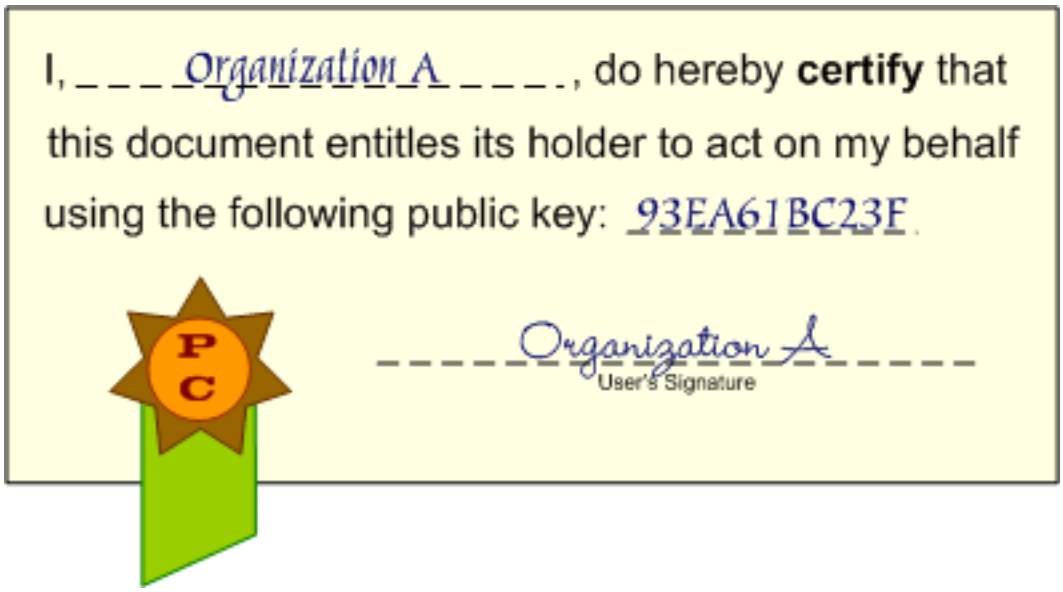
- **Turn down B's request** . C doesn't trust B, and that's that.
- **Accept B's request** . The 'original' requestor is A so, although C is answering a request from B, it will actually be carrying out a job for A.

In this situation, it seems logical that C should *accept* B's request. However, C has to know that B's request is performed on behalf of A:



Of course, this is not a very secure solution, since *anyone* could claim to be acting on A's behalf! One possible solution would be for C to contact A every time it receives a request on A'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 B. Organization A would be flooded with messages saying "B just asked me to perform a task on your behalf... can you confirm that this is correct?". In response, A would have to mutually authenticate itself with all those organizations and give a confirmation.

A more elegant solution would be to somehow make Organization C believe that Organization B *is* Organization A. In other words, it would interesting to find a legitimate way for B to demonstrate that it is, in fact, acting on A's behalf. One way of doing this would be for A to 'lend' its public and private key pair to B. 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 something like this:

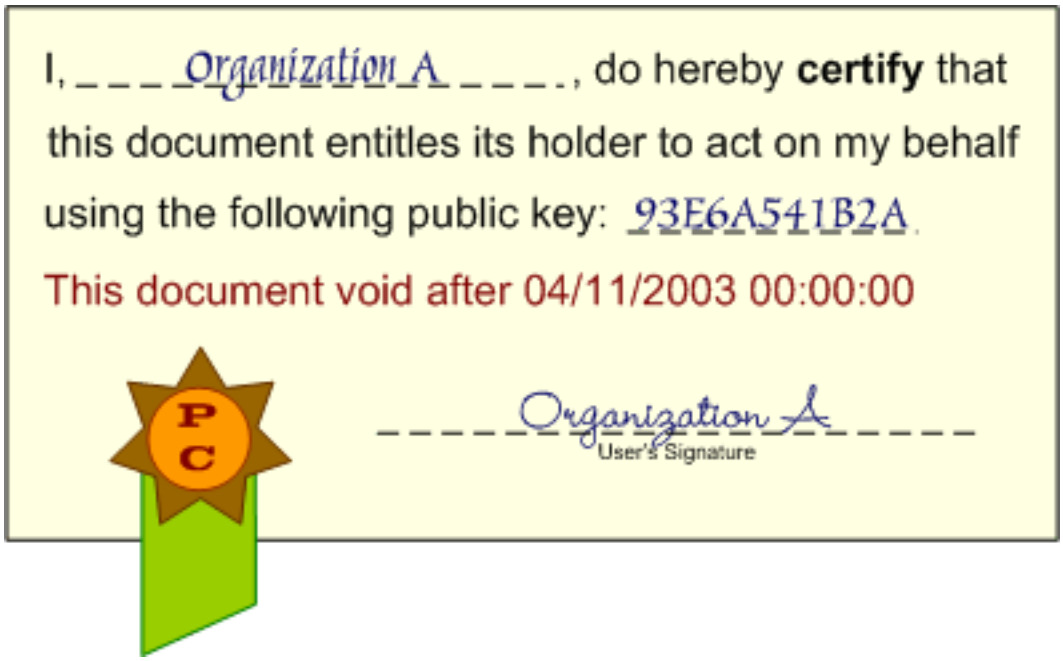


The solution: proxy certificates

The certificate pictured above 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 the previous chapter, 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 the previous chapter).

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 described shortly.

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 24 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 following:



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 mutually 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 mutual 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). We're not going to discuss all the added benefits of proxy certificates, since in a Grid Services-based application we will be mainly concerned with delegation and single sign-on. However, a link is provided at the end of this page if you are interested in reading more about proxy certificates.

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 I say "Proxy certificates are really nifty!", and are not totally convinced that they are secure, the following paragraphs give a much more detailed look at the process of creation and validation of a proxy certificate. However, the rest of this page can be safely skipped 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 the first picture) 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.

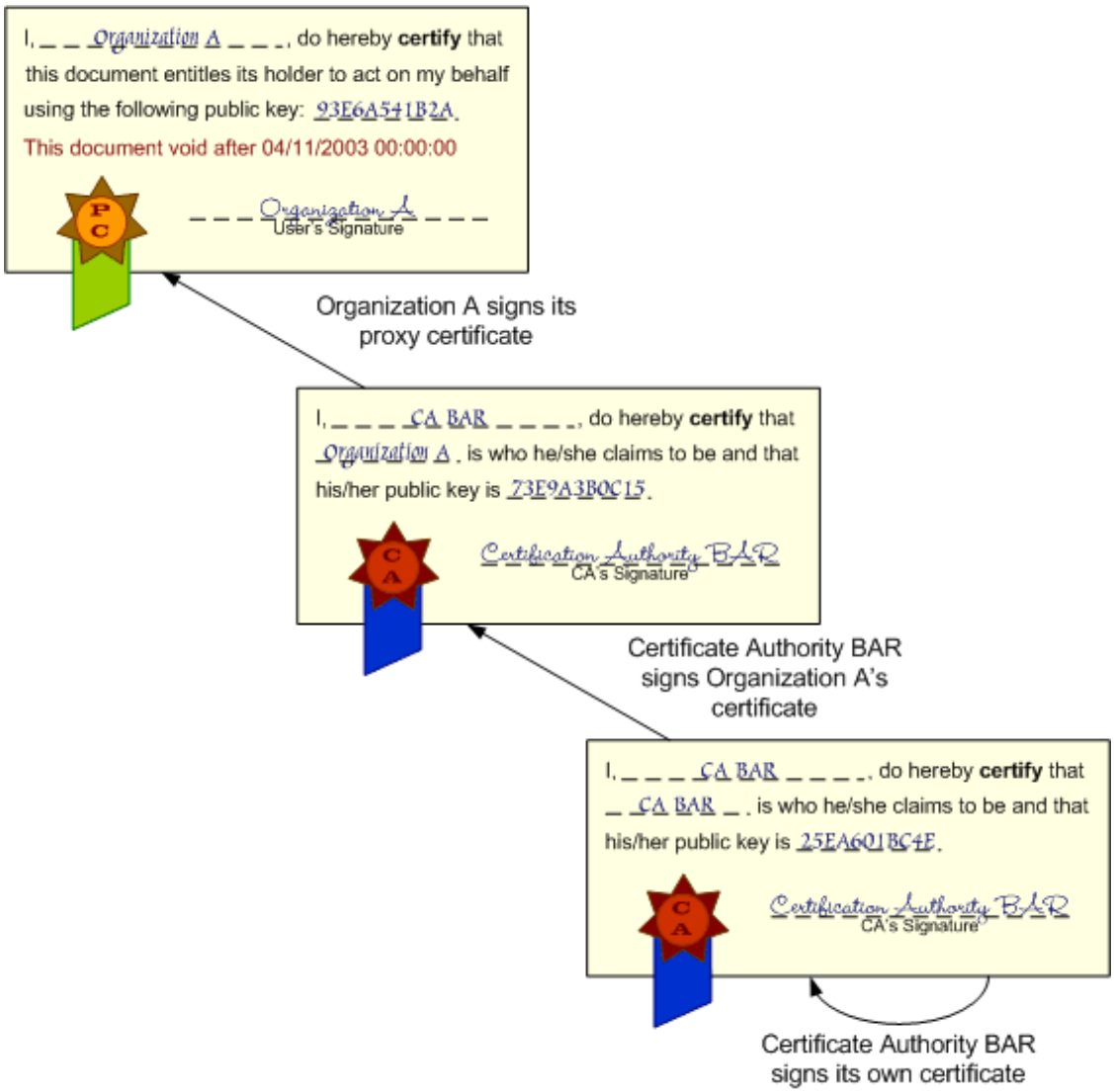
Notice how the proxy's private key is never transmitted between A and B. This is also true of A's private key.

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 previous chapter. 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.

The following illustration shows the chain of signatures that we could find in a proxy certificate:



That's (not) all, folks!

There's a lot more to proxy certificates than what has been explained in this page. For example, you can use proxy certificates to sign other proxy certificates. However, for the tutorial, the material covered in this page should be enough. If you want to take a closer look at proxy certificates, and everything that can be done with them, I highly recommend reading the following Internet Draft: Internet X.509 Public Key Infrastructure Proxy Certificate Profile [http://www.ietf.org/internet-drafts/draft-ietf-pkix-proxy-10.txt].

Authorization types

Finally, we're going to describe the different *authorization* types in GSI. Remember that, although authorization is not one of the 'fundamental pillars' of a secure conversation, it is nonetheless a very important concept. Authorization refers to who is *authorized* to perform a certain task. In a Grid Services context, we will generally need to know who is authorized to use a certain grid service.

GSI supports authorization in both the server-side and the client-side. Each have three authorization mechanisms, and we will be able to choose one of them when we start writing secure grid services.

Server-side authorization

The server has three authorization modes. Depending on the authorization mode we choose, the server will decide if it accepts or declines an incoming invocation.

- **None**: This is the simplest type of authorization. No authorization will be performed.
- **Self**: A client will be allowed to use a grid 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). We will see them in detail later on. When this type of authorization is used, only the users that are listed in the service's gridmap may invoke it.

Client-side authorization

This allows the client to figure out when it will allow a grid 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. If we use both client-side and server-side Self authorization, a service can be invoked *if and only if* its identity matches the client's.
- **Host**: The client will authorize an invocation if the host returns an identity containing the hostname. This is done using *host certificates*. The tutorial currently doesn't cover host certificates, although a future version will.

Please note that it's very easy to mistake client- and server-side authorization because they have two mechanisms in common.

Chapter 12. Setting up GSI

Up to this point, we've reviewed a couple of fundamental security concepts, and all the features of the Grid Security Infrastructure (GSI). So, we should be ready to start churning out secure services, right? Well, we're not quite there yet. Remember that authentication is a very important concern in Grid applications and that, to perform strong authentication, a digital certificate is a must. If you're just starting with Globus and Grid programming, chances are you don't have one. So, before we can start writing secure services, we need to get one!

Don't worry, no purchase necessary. We won't be going to one of those big and expensive Certificate Authority, but instead will be installing a very simple CA developed by the folks a Globus. To be precise, we will be doing the following:

1. Create a set of UNIX users in our computer.
2. Install a simple Certificate Authority.
3. Create digital certificates for the users, using the CA installed in our computer.

Creating users

To run the secure examples in a slightly realistic fashion, we will need to have two users created in our machine. Actually, the first one is (quite probably) already there: you user account. In my case, this would be the `borja` UNIX account. This account will be used to run the client programs.

Later on, we will give this user with a digital certificate with the following distinguished name:

```
O=Globus, OU=GT3 Tutorial, CN=Borja Sotomayor
```

Of course, you can change the values of the distinguished name to match your organization, organizational unit, and common name.

The second user you need to create is a generic `globus` account which will be used to perform administrative tasks such as starting and stopping the container, deploying services, etc. This user will also be in charge of managing the simple CA we are going to install. To be able to do this, make sure this account has read and write permissions in the `$GLOBUS_LOCATION` directory.

Later on, we will give this user a digital certificate with the following name:

```
O=Globus, OU=GT3 Tutorial, CN=Globus 3 Administrator
```

It's quite possible that you already have a separate `globus` account for this, since it is commonplace in UNIX systems to create generic accounts to run specific services (the `www-data` account, the `proxy` account, etc.) However, you might also be using this account to run the client programs. If so, from now on, you should use the `globus` account only for administrative tasks, and your user account (`borja`, in my case) to run the clients.

Why it is unwise to run the container and the cli-

ents with the same user

At this point, you might be thinking: "Sure, having two separate users seems like the right thing to do, but this is just a tutorial... I guess I can work with just one user". If this is so, vanquish that thought from your head immediately. It is certainly possible to work through all the security examples using just one user (for both the container and the client programs), but doing so might 'mask' errors and pitfalls which might reveal themselves when you run the examples in a more real (and less tutorial-like) situation: having the container on one machine, running under a certain identity, and having the client programs running in a different machine under a completely different identity.

Bottom line: please do take the time to create a `globus` account to run the container and the Certificate Authority, and a user account to run the example client applications.

Installing SimpleCA

We are now going to install a very simple Certificate Authority, appropriately called *SimpleCA*, developed by the folks at Globus. We'll use SimpleCA to create certificates for the globus account and our user account. Bear in mind that, although it is very easy to use, it might not be appropriate for production systems. Consider OpenCA [http://www.openca.org/] as a more powerful alternative. Also, in case you don't want to install a root CA in your organization, and use a more widely known one, you can find a list of academic CAs here [http://www.terena.nl/tech/task-forces/tf-aace/tacar/].

However, bear in mind that the tutorial is written with SimpleCA in mind. Even so, you should have no problem switching to a different CA once you've completed the examples and understood how GT3 manages certificates.

Download SimpleCA

First of all, you need to download SimpleCA. You can do so here [http://www.globus.org/security/simple-ca.html].

Building SimpleCA

Once you've downloaded the SimpleCA installation package (a file called globus_simple_ca-latest-src_bundle.tar.gz), you have to run the following command from the directory where you've downloaded the file:

```
$GLOBUS_LOCATION/sbin/gpt-build globus_simple_ca-latest-src_bundle.tar.gz gcc32dbg
```

This builds the SimpleCA package, and outputs a couple of status messages. Don't worry if you see the following message: `make: *** No rule to make target `distclean'. Stop.`. It is perfectly normal.

Setting up SimpleCA

Now that SimpleCA has been built, we need to run a post-install script to set it up. Just run the following command:

```
$GLOBUS_LOCATION/sbin/gpt-postinstall
```

You'll should see the following output:

```
    C e r t i f i c a t e   A u t h o r i t y   S e t u p

This script will setup a Certificate Authority for signing Globus
users certificates.  It will also generate a simple CA package
that can be distributed to the users of the CA.

The CA information about the certificates it distrubtes will
be kept in:

$GLOBUS_USER_HOME/.globus/simpleCA/

The unique subject name for this CA is:

cn=Globus Simple CA, ou=simpleCA-localhost, ou=GlobusTest, o=Grid

Do you want to keep this as the CA subject (y/n) [y]:
```

The script is asking us to define the subject that will appear in the CA's digital certificate. Remember that this is a special self-signed certificate which will identify the CA, and which can be used to verify the validity of certificates signed by the CA. Although you can certainly keep the default subject name, we're going to change it. Answer 'n' to the question. You should now see this:

```
Enter a unique subject name for this CA:
```

We will use the following subject name:

```
cn=Globus Simple CA, ou=GT3 Tutorial, o=Globus
```

Now you should see the following:

```
Enter the email of the CA (this is the email where certificate
requests will be sent to be signed by the CA):
```

You can enter any e-mail you want, since we're not actually going to use it. When we start requesting certificates, this is the e-mail we're supposed to send the certificate request to. However, since we're working on a single machine, we'll be able to do the whole process in the comfort of our own hard disk.

Once you've entered the e-mail address, you should see the following:

```
The CA certificate has an expiration date. Keep in mind that
once the CA certificate has expired, all the certificates
signed by that CA become invalid.  A CA should regenerate
the CA certificate and start re-issuing ca-setup packages
before the actual CA certificate expires.  This can be done
by re-running this setup script.  Enter the number of DAYS
the CA certificate should last before it expires.
[default: 5 years (1825 days)]
```

We won't be too concerned about the expiration date of the CA certificate, so we can safely press enter here to select the default value (5 years).

Now, the install script will start generating the certificate:

```
Using configuration from /home/globus/.globus/simpleCA//grid-ca-ssl.conf
Generating a 1024 bit RSA private key
...........................++++++
.............++++++
writing new private key to '/home/globus/.globus/simpleCA//private/cakey.pem'
```

However, when the install script comes to the point where it must generate the certificate's *private key*, it will ask you for a password. Remember, the private key must be known only by the certificate's owner (in this case, the CA), and what better way to ensure this than by protecting it with a password.

```
Enter PEM pass phrase:
```

Enter any password. To avoid confusion with other password we will be using, I suggest you simply enter the following password: simpleca. However, if you plan to use this CA in a production environment, feel free to enter any password. You will be asked to repeat it:

```
Verifying password - Enter PEM pass phrase:
```

Any time we need to access the CA's private key, we will need to provide this password. For example, since the private key is needed to digitally sign certificates, we'll need to provide the password each time the CA issues a certificate.

After you enter the password and confirm it, you will be asked no more questions. You will see a rather lengthy output which you can safely ignore. However, let's take a look a close look at some particular messages which basically confirm that we've successfully set up a CA

First off, take a look at this:

```
A self-signed certificate has been generated
for the Certificate Authority with the subject:

/O=Globus/OU=GT3 Tutorial/CN=Globus Simple CA

If this is invalid, rerun this script

$GLOBUS_LOCATION/setup/globus/setup-simple-ca

and enter the appropriate fields.

-----------------------------------------------------------------

The private key of the CA is stored in $GLOBUS_USER_HOME/.globus/simpleCA//private
The public CA certificate is stored in $GLOBUS_USER_HOME/.globus/simpleCA//cacert.
```

This message confirms that the CA's certificate has, in fact, been created. We are also told where the certificate can be found, along with the private key. If you try to open the certificate, you'll see that its contents look like gibberish. If you want to take a peek at all the values it contains, you can use a very handy tool included with the toolkit called grid-cert-info:

```
grid-cert-info -file ~/.globus/simpleCA/cacert.pem
```

You will also see the following message in the final output, which tell us that the setup isn't quite com-

plete yet:

```
Note: To complete setup of the GSI software you need to run the
following script as root to configure your security configuration
directory:

$GLOBUS_LOCATION/setup/globus_simple_ca_24d355a5_setup/setup-gsi
```

You should also see the same message telling you that you should run $GLOBUS_LOCATION/setup/globus/setup-gsi to complete setup. We'll get to that in a second. The setup-gsi finishes the setup of GSI on our system. To do this, it creates a set of configuration files in the /etc directory, so this command should be run as root. However, in systems without root access, you can use a -noroot argument to specify an alternate location which is non-root writable. Let's suppose you do have root access, and run the command:

```
$GLOBUS_LOCATION/setup/globus_simple_ca_24d355a5_setup/setup-gsi
```

You should see the following:

```
   G S I   :   C O N F I G U R A T I O N   P R O C E D U R E


Before you use the Grid Security Infrastructure, you should first
define the DN (distinguished name) that should be used for your
organization's X509 certificates.  If you do not define a DN,
a default DN will be assigned to you.

This script will ask some questions about site specific
information. This information is used to configure
the Grid Security Infrastructure for your site.

For some questions, a default response is given in [].
Pressing RETURN in response to such a question will enable the default.
This script will overwrite the file --

     /etc/grid-security/certificates//grid-security.conf.24d355a5


Do you wish to continue (y/n) [y] :
```

Answer yes. You should now see the following:

```
=========================================================================

(1) Base DN for user certificates
         [ ou=,  ou=GT3 Tutorial, o=Globus ]
(2) Base DN for host certificates
         [  ou=GT3 Tutorial, o=Globus ]

=========================================================================
(q) save, configure the GSI and Quit
(c) Cancel (exit without saving or configuring)
(h) Help
=========================================================================
```

The script asks us to provide a distinguished name which will be the base for user and host certificates (issued by the CA). Although we can change these names, we'll simply accept the default ones. Notice how the OU (Organizational Unit) and O (Organization) are the same as the CA's OU and O (specified earlier while installing SimpleCA). To continue, answer 'q'.

We are very nearly finished setting up SimpleCA. Before moving on to the very last step, remember there was also a message asking you to run /usr/local/gt3/setup/globus/setup-gsi. We won't be running this script because its purpose is to enable our system to work with the Globus CA, which is no longer active. We will only be working with our SimpleCA.

Setting up the default CA

The only thing left to do is to configure our system so that it will use the SimpleCA we just installed as the default CA. We can do this simply by running the following from the root account:

```
$GLOBUS_LOCATION/bin/grid-default-ca
```

You should see the following:

```
The available CA configurations installed on this host are:

1) 24d355a5 -  /O=Globus/OU=GT3 Tutorial/CN=Globus Simple CA
/bin/ls: /etc/grid-security//grid-security.conf: No such file or directory

The default CA is:


Enter the index number of the CA to set as the default:
```

Note

Don't worry about the `No such file or directory` message. It's due to the fact that we haven't chosen a default CA yet. Type in 1 to make our SimpleCA the default CA.

Summing up...

Where do we stand right now? Well, we now have a fully functional CA working on our computer (Yay!). This Certificate Authority is managed by the `globus` user. We are now ready to request certificates to our CA. We can do this from the machine that hosts the CA, or from other machines. If you are going to request a certificate from a different machine, you'll need to read the following page. Otherwise, you can safely skip it.

Installing the CA Distribution Package

You only need to read this page if you are planning on requesting certificates to the SimpleCA we just installed *from another machine*. If you are going to run all the examples from one machine, you can safely skip this page.

To request certificate from another machine, we need to install our CA's distribution package in that machine. This package is creating during the CA's installation. In fact, let's take a look at another message which is output at the end of the installation procedure:

```
The distribution package built for this CA is stored in

$GLOBUS_USER_HOME/.globus/simpleCA//globus_simple_ca_24d355a5_setup-0.13.tar.gz

This file must be distributed to any host wishing to request
certificates from this CA.
```

That file is the distribution package (please note that the hash code, 24d355a5, will probably be different in your machine). You need to copy that file to the machine where you'll be making the requests and, using a user with write permissions in `$GLOBUS_LOCATION`, run the following:

```
$GLOBUS_LOCATION/sbin/gpt-build globus_simple_ca_HASH_setup-0.13.tar.gz
```

Once the distribution package has been built, you'll need to run the setup script to complete the installation:

```
$GLOBUS_LOCATION/setup/globus_simple_ca_HASH_setup/setup-gsi
```

Requesting a certificate

We are now going to request a certificate from the SimpleCA for our user account (in my case, `borja`). Remember, this is the account we'll use to run all the client applications.

From the user account, run the following command:

```
$GLOBUS_LOCATION/bin/grid-cert-request
```

You should see the following output:

```
A certificate request and private key is being created.
You will be asked to enter a PEM pass phrase.
This pass phrase is akin to your account password,
and is used to protect your key file.
If you forget your pass phrase, you will need to
obtain a new certificate.

Using configuration from /etc/grid-security/globus-user-ssl.conf
Generating a 1024 bit RSA private key
..............................++++++
.......................................++++++
writing new private key to '/home/borja/.globus/userkey.pem'
Enter PEM pass phrase:
Verifying password - Enter PEM pass phrase:
```

You are being asked for a password to protect your private key. Don't confuse this password with the one we provided when configuring SimpleCA. That password protects the CA's private key. The one we're being asked for now will protect our user account's private key. We will need this password each time we want to access our certificate's private key. For example, we will need it when generating proxy certificates (since our private key is required to digitally sign the proxy certificate). To avoid confusion with other passwords we'll be using in the tutorial, and if you're not going to use this certificate for anything but the tutorial, I suggest you simply enter your username as the password (in my case, `borja`).

After you enter and confirm the password, you should see the following output:

```
A private key and a certificate request has been generated with the subject:

/O=Globus/OU=GT3 Tutorial/CN=Borja Sotomayor

If the CN=Borja Sotomayor is not appropriate, rerun this
script with the -force -cn "Common Name" options.

Your private key is stored in /home/borja/.globus/userkey.pem
Your request is stored in /home/borja/.globus/usercert_request.pem

Please e-mail the request to the Globus Simple CA borja@borjanet.com
You may use a command similar to the following:

  cat /home/borja/.globus/usercert_request.pem | mail $EMAIL_ADDRESS

Only use the above if this machine can send AND receive e-mail. if not, please
mail using some other method.

Your certificate will be mailed to you within two working days.
If you receive no response, contact Globus Simple CA at borja@borjanet.com
```

A *certificate request* has been generated and placed in the `$HOME/.globus/usercert_request.pem`. This certificate request has to be signed by a CA so it will be a complete digital certificate. We will do this in the next page.

Signing the certificate with SimpleCA

We need to send this request to our CA so it can digitally sign it. Although the grid-cert-request asks you to mail it to the CA's email address, we won't need to do this procedure through email since the CA and the requesting user are in the same machine. We'll be able to 'send' the request the CA through a temp directory in our hard disk. However, bear in mind that, in real applications, it is commonplace to send the requests to a CA administrator through email.

Using your user account, do the following:

```
cp $HOME/.globus/usercert_request.pem /tmp
```

Now, using the `globus` account, do the following:

```
$GLOBUS_LOCATION/bin/grid-ca-sign -in /tmp/usercert_request.pem -out /tmp/usercert
```

The grid-ca-sign command is used to sign certificate requests and generate valid certificates. Since this operation needs the CA's private key (to sign the certificate), we'll need to enter the CA's password:

```
Enter password for the CA key:
```

You should now see the following output:

```
The new signed certificate is at: $GLOBUS_USER_HOME/.globus/simpleCA//newcerts/01.
```

A digital certificate has been generated from the request and has been deposited in `/tmp/usercert.pem`. However, CA always keeps a copy of its certificates. Now, all we have to do is retrieve the certificate from the `/tmp` directory (using our user account):

```
cp /tmp/usercert.pem $HOME_DIR/.globus/usercert.pem
```

Voilà! You are now a fully certified user!

Bear in mind that it is very important to respect the directory name (`$HOME_DIR/.globus`) and the file name (`usercert.pem`) when installing the new certificate. Otherwise, any application and utility which needs to use the certificate will fail to find it and, therefore, not work.

Finally, remember that the procedure we've followed to request and sign the certificate has been a bit atypical since both the CA and the requestor are in the same machine. A more 'real' procedure would be the following:

1. User A creates a certificate request.
2. User A sends the certificate request to a CA via email (e.g. ca-admin@foobar.com)
3. The CA's administrator receives the request, reviews it, and decides if it will be approved. If the request is approved, the CA administrator signs the request using the CA's private key, and sends the certificate to User A via email (e.g. user-a@somecompany.com)
4. User A receives the certificate and installs it in the `$HOME/.globus directory`.

Final steps

Although our user account has a certificate, we're not quite done yet. Just two final steps to do before we can (finally) start writing secure grid services.

Requesting a certificate for the `globus` account

The `globus` account also needs its own certificate, which it will use to run the container and its services. Don't confuse the `globus` user's certificate with the CA's certificate (which we created when setting up SimpleCA, using the `globus` account). At this point, the `globus` account *does not* have a certificate, it only manages a CA which has a self-signed certificate.

Requesting and signing a certificate for the `globus` account is very simple, since the steps you have to follow are exactly the same as the ones described in the previous two pages.

Creating proxy certificates

An operation we'll be performing frequently during the security examples is creating proxy certificates. So, before we move on to the examples, let's make sure we can successfully create proxy certificates.

First of all, run the following from your user account:

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

This sets up a couple of environment variables. Now, run the following command:

```
grid-proxy-init
```

You should see the following output:

```
Your identity: /O=Globus/OU=GT3 Tutorial/CN=Borja Sotomayor
Enter GRID pass phrase for this identity:
```

Enter the password which protects your user's private key. You should now see the following:

```
Creating proxy ..................................................... Done
Your proxy is valid until: Sun Feb 15 22:55:45 2004
```

The date should be 12 hours more than the current time. If you are of a curious disposition, you can take a look at some of the proxy certificate's contents by running the following command:

```
grid-proxy-info
```

This command will show, among other things, the path were the proxy certificate has been created. You can take an even closer look at the contents of the proxy certificate by running the following:

```
grid-cert-info -file $PATH_TO_PROXY_CERT
```

Now, repeat all these steps for the `globus` account to test if you can create a proxy certificate for that account.

Chapter 13. Writing a Secure Math Service

We are now all set to enter the secure grid services nirvana. We have a CA and we have the certificates. We're finally ready to create our very first secure grid service! After all the work required to setup the CA and create the certificate, you might be thinking that enabling a grid service for security must be terribly complicated. Well, you'll be happy to know it's not! The hardest part if getting the CA and the certificates to work properly.

Note

Did you read the Introduction to the GT3 Security Services part of the tutorial? (titled "Before reading this part of the tutorial...") If the answer is 'no', please take the time to read it (it's only two paragraphs...)

Are we all set? Ok, let's go!

A secure service

The service interface

The interface for our secure grid service is just our ordinary everyday `add`, `subtract`, and `getValue` from the first chapters of the previous part (GT3 Core).

Note

The GWSDL file for this example can be found here: `$TUTORIAL_DIR/schema/progtutorial/MathService/Math.gwsdl`

There is no need to create a new GWSDL file since adding security to a service doesn't affect the interface description (i.e. the GWSDL file)

The service implementation

We will be using an OperationProvider to implement our service. This operation provider will include the `add`, `subtract`, and `getValue` implementations, callback methods (although we'll only implement `postCreate`), and an additional private method `logSecurityInfo` which will log certain security information. In this example, the information logged by this method won't be very useful, but it will be relevant in future examples.

Note

The full code for the OperationProvider can be found in `$TUTORIAL_DIR/org/globus/progtutorial/services/security/first/impl/MathProvider.java`

We'll now take a close look at the more relevant parts of the code.

The implementation of the public methods is very simple. Notice how we're calling logSecurityInfo in all of them:

```
public void add(int a) throws RemoteException
{
  logSecurityInfo("add");
  value = value + a;
}

public void subtract(int a) throws RemoteException
{
  logSecurityInfo("subtract");
  value = value - a;
}

public int getValue() throws RemoteException
{
  logSecurityInfo("getValue");
  return value;
}
```

The postCreate callback method simply calls logSecurityInfo:

```
public void postCreate(GridContext context) throws GridServiceException
{
  logSecurityInfo("postCreate");
}
```

Finally, the logSecurityInfo writes some security information to the container's log. As mentioned earlier, this code won't have any relevance until we move on to the following examples. In fact, we won't be paying special attention to any of this code (except the part where it writes the caller's identity to the log, highlighted in bold)

```
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 the caller's subject
  subject = JaasSubject.getCurrentSubject();
  logger.info("INVOCATION SUBJECT");
  logger.info(subject==null?"NULL":subject.toString());

  // Print out service subject
  logger.info("SERVICE SUBJECT");
  subject = SecurityManager.getManager().getServiceSubject(base);
  logger.info(subject==null?"NULL":subject.toString());

  // Print out system subject
  logger.info("SYSTEM SUBJECT");
  try{
    subject = SecurityManager.getManager().getSystemSubject();
```

```
    logger.info(subject==null?"NULL":subject.toString());
  }catch(Exception e)
  {
      logger.warn("Unable to obtain service subject");
  }
}
```

Notice how enabling security in a grid service doesn't affect the server-side code at all (at least at this point; more complicated security scenarios will require that we add code on the server-side).

Deployment descriptor parameters

To tell the container that our service is going to be a secure service, we need to add two parameters to the deployment descriptor:

- The security configuration file (`securityConfig` parameter)
- Authorization method to use (`authorization` parameter)

The securityConfig parameter

This parameter tells the container where it can find the *security configuration file*. This is an XML file that includes configuration details related to security, such as what level of security is required when invoking certain methods. For now, we are going to use a default security configuration file included with GT3. This default file specifies that *all* the methods must be accessed using a secure conversation. We will see how we can write our own security configuration file in the next section.

To tell the container to use the default configuration file for our service, we'll need to add the following lines to the deployment descriptor:

```
<parameter name="securityConfig"
  value="org/globus/ogsa/impl/security/descriptor/gsi-security-config.xml"/>
```

The authorization parameter

This parameters allows us to specify the type of server-side GSI authorization to use in this service. For now, we'll use the simplest type of authorization: no authorization. We'll look into other authorization types in the following sections.

To use no authorization, we'll need to add the following lines to the deployment descriptor:

```
<parameter name="authorization" value="none"/>
```

The full deployment descriptor

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

  <service name="progtutorial/security/first/MathService" provider="Handler" style
    <parameter name="name" value="Secure MathService"/>
    <parameter name="schemaPath" value="schema/progtutorial/MathService/Math_servi
    <parameter name="className" value="org.globus.progtutorial.stubs.MathService.M
    <parameter name="operationProviders"
      value="org.globus.progtutorial.services.security.first.impl.MathProvider"/>
    <parameter name="baseClassName" value="org.globus.ogsa.impl.ogsi.GridServiceIm

    <parameter name="securityConfig"
      value="org/globus/ogsa/impl/security/descriptor/gsi-security-config.xml"/>
    <parameter name="authorization" value="none"/>

    <!-- Start common parameters -->
    <parameter name="allowedMethods" value="*"/>
    <parameter name="persistent" value="true"/>
    <parameter name="handlerClass" value="org.globus.ogsa.handlers.RPCURIProvider"
  </service>
</deployment>
```

Note

This file is `$TUTORIAL_DIR/org/globus/progtutorial/services/security/first/server-deploy.wsdd`. You'll notice this WSDD file actually has information about more services. These services are explained in the next section.

A secure client

The client used to invoke the secure service will be almost identical to clients seen in the previous part of the tutorial (GT3 Core). In fact, to configure our client for a basic secure invocation we only need to add two lines!

```
((Stub)math)._setProperty(Constants.GSI_SEC_CONV,Constants.ENCRYPTION);
((Stub)math)._setProperty(Constants.AUTHORIZATION,NoAuthorization.getInstance());
```

What these two lines do is configure the stub class (a MathPortType object) to use security. More specifically, we're doing the following:

- We're telling the stub to use full-blown encryption. GSI allows us to choose between different levels of security, and encryption is just one of them. For example, we could have chosen to guarantee message integrity using a digital signature (without encrypting the whole message). All the stub options are described in the Stub security options appendix.
- 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 GSI authorization page to refresh your memory.

As mentioned before, besides those two lines, the rest of the client is practically identical to the previous ones seen in the tutorial. The only difference is that we'll be putting the `add`, `subtract`, and `getValue` calls inside `try...catch` blocks to observe how certain exceptions are raised in certain circumstances (we'll see this in the following sections).

```
package org.globus.progtutorial.clients.MathService;

import org.globus.progtutorial.stubs.MathService.service.MathServiceGridLocator;
import org.globus.progtutorial.stubs.MathService.MathPortType;

import org.globus.ogsa.impl.security.Constants;
import org.globus.ogsa.impl.security.authorization.NoAuthorization;

import javax.xml.rpc.Stub;
import java.net.URL;

public class ClientGSIConvEncrypt
{
  public static void main(String[] args)
  {
    try
    {
      // Get command-line arguments
      URL GSH = new java.net.URL(args[0]);
      int a = Integer.parseInt(args[1]);

      // Get a reference to the MathService instance
      MathServiceGridLocator mathLocator =   new MathServiceGridLocator();
      MathPortType math = mathLocator.getMathServicePort(GSH);

      // Setup security options
      ((Stub)math)._setProperty(Constants.GSI_SEC_CONV,Constants.ENCRYPTION);
      ((Stub)math)._setProperty(Constants.AUTHORIZATION,NoAuthorization.getInstanc


      // Call remote method 'add'
      try{
        math.add(a);
        System.out.println("Added " + a);
      } catch(Exception e)
      {
        System.out.println("ERROR: " + e.getMessage());
      }

      // Call remote method 'subtract'
      try{
        math.subtract(1);
        System.out.println("Subtracted 1");
      } catch(Exception e)
      {
        System.out.println("ERROR: " + e.getMessage());
      }

      // Get current value through remote method 'getValue'
      try{
        int value = math.getValue();
        System.out.println("Current value: " + value);
      } catch(Exception e)
      {
        System.out.println("ERROR: " + e.getMessage());
      }
    }catch(Exception e)
    {
```

```
      System.out.println("ERROR!");
      e.printStackTrace();
    }
  }
}
```

Note

This file is `$TUTORIAL_DIR/org/globus/progtutorial/clients/MathService/ClientGSIConvEncrypt.java`

Let's give it a try...

Ok, we're now set to give this a try.

Compile and deploy

First of all, we'll need to build the service:

```
./tutorial_build.sh \
org/globus/progtutorial/services/security/first \
schema/progtutorial/MathService/Math.gwsdl
```

Now, we have to deploy it. Remember that you should do this from the globus account:

```
ant deploy \
-Dgar.name=$TUTORIAL_DIR/build/lib/org_globus_progtutorial_services_security_first
```

Activating logging

Since we are using the logging classes in the service, we'll need to activate logging for our service. Add the following line at the end of `$GLOBUS_LOCATION/ogsilogging.properties`:

```
org.globus.progtutorial.services.security.first.impl.MathProvider=console,info
```

Starting the container

Before we start the container, make sure you've created a proxy certificate for the `globus` account (this procedure was described in this page). We need to create a proxy certificate because the default behavior in GT3 is to use the proxy certificate for authentication. Of course, we can also configure the container to directly use other certificates, but the tutorial currently doesn't cover that (although it eventually will).

Once you've created the proxy certificate, start the container (using the `globus` account):

```
globus-start-container
```

Compiling the client

Let's compile the client:

```
javac \
-classpath ./build/classes/:$CLASSPATH \
org/globus/progtutorial/clients/MathService/ClientGSIConvEncrypt.java
```

Before running any of the client applications, we also need to create a proxy certificate for our user account (in my case, `borja`). Again, the default behavior in the client-side is to use a proxy certificate for authentication, so we need to create one first.

Now, run the client:

```
java \
-classpath ./build/classes/:$CLASSPATH \
org/globus/progtutorial/clients/MathService/ClientGSIConvEncrypt \
http://127.0.0.1:8080/ogsa/services/progtutorial/security/first/MathService \
5
```

If all goes well, you should see this in the client side:

```
Added 5
Subtracted 1
Current value: 4
```

And the following on the server side:

```
INFO: SECURITY INFO FOR METHOD 'add'
INFO: The caller is:/O=Globus/OU=GT3 Tutorial/CN=Borja Sotomayor

INFO: INVOCATION SUBJECT
INFO: Subject:
        Principal: /O=Globus/OU=GT3 Tutorial/CN=Globus 3 Administrator
        Private credential: org.globus.gsi.gssapi.GlobusGSSCredentialImpl@ae1393

INFO: SERVICE SUBJECT
INFO: NULL

INFO: SYSTEM SUBJECT
INFO: Subject:
        Principal: /O=Globus/OU=GT3 Tutorial/CN=Globus 3 Administrator
        Private credential: org.globus.gsi.gssapi.GlobusGSSCredentialImpl@ae1393
```

Notice how the caller's subject is the one in my account's certificate while the invocation and system subject is the subject of the certificate belonging to the `globus` account.

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?"

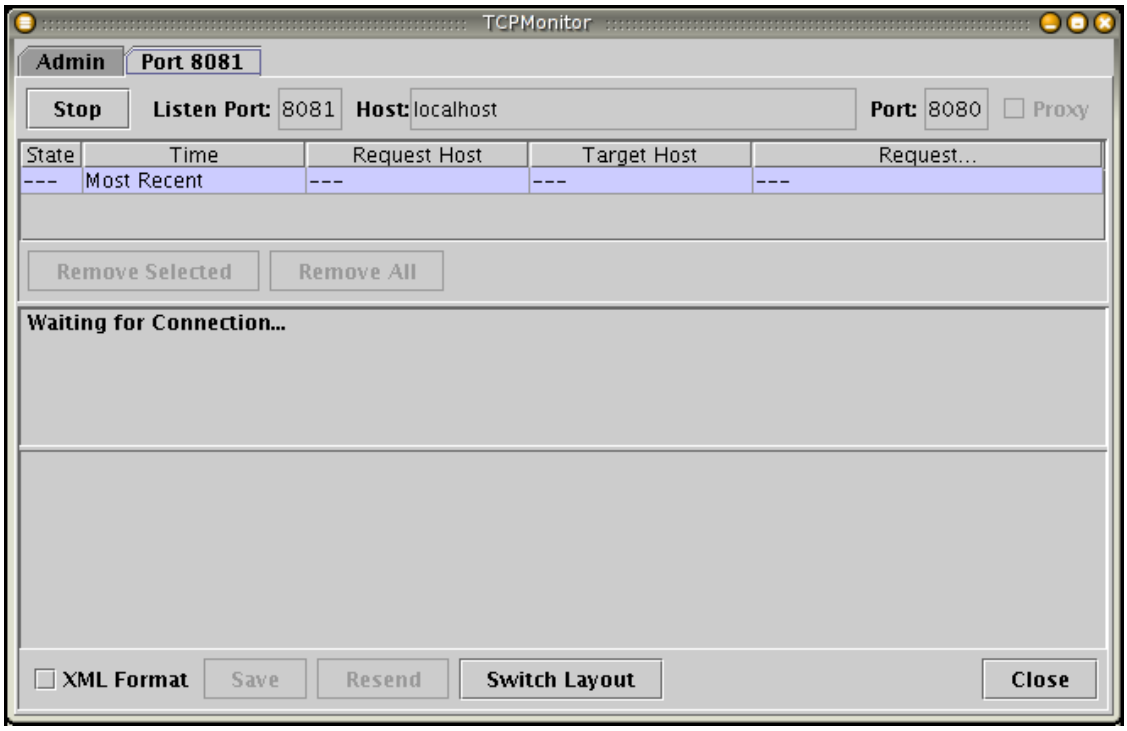
To empirically prove that it is doing the 'encryption thingy', we are going to use a handy-dandy command-line tool included with Axis called TCPMonitor. This tool allows us to intercept the data that is sent from the client to the server (and vice versa). We will see how the information is, in fact, encrypted.

To start TCPMonitor, run this:

```
java org.apache.axis.utils.tcpmon 8081 localhost 8080
```

This starts an instance of the TCPMonitor. 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.

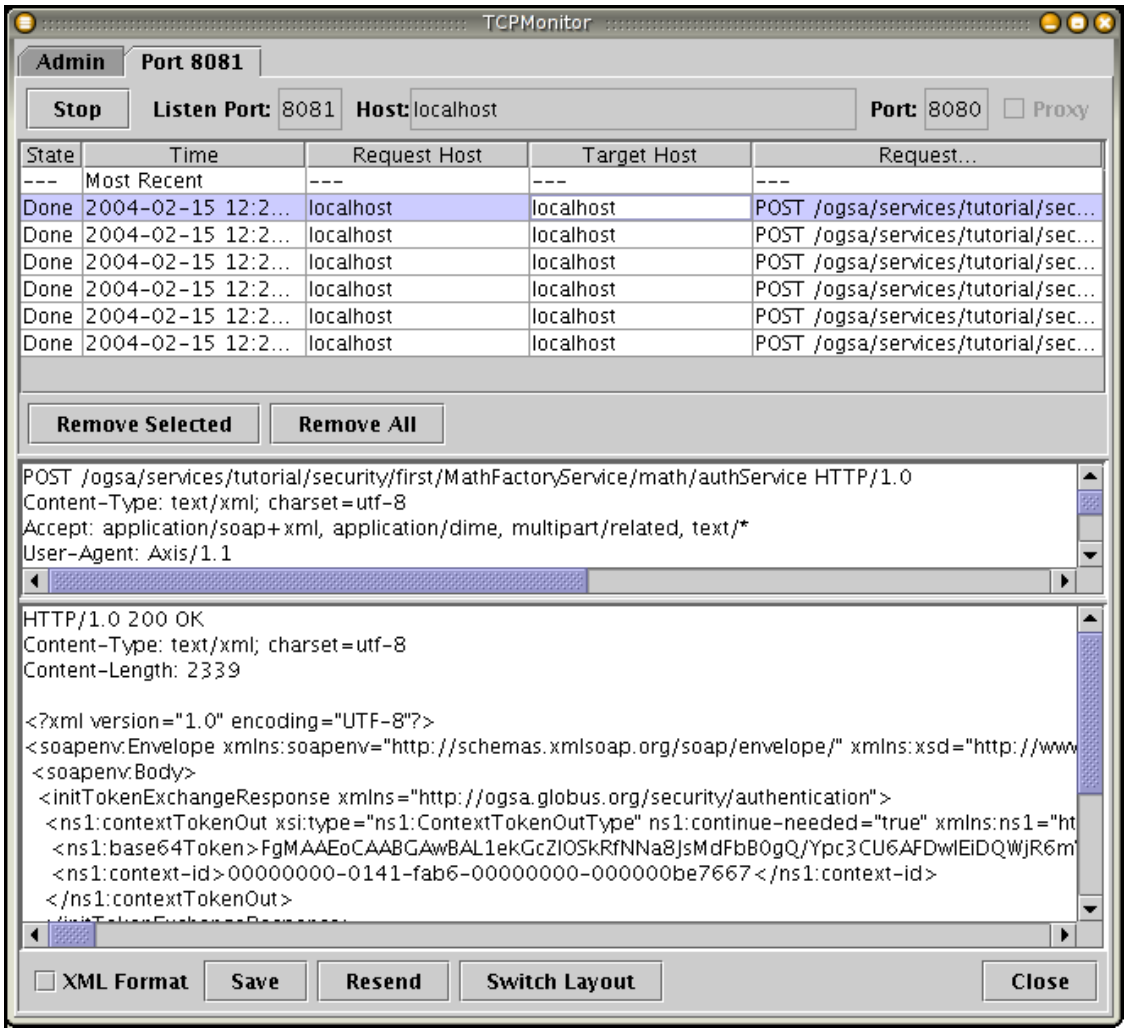
The TCPMonitor interface should look something like this:



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/classes/:$CLASSPATH \
org/globus/progtutorial/clients/MathService/ClientGSIConvEncrypt \
http://127.0.0.1:8081/ogsa/services/progtutorial/security/first/MathService \
5
```

Once you've invoked the service, you should see the following in the TCPMonitor:



The top list shows all the connections 'intercepted' by the TCPMonitor. You can select any of them to see what was sent to the server (first text area) and what the server replied to the client (second text area). There should be three pairs of connections (on pair for each of the invocations the client makes: `add`, `subtract`, and `getValue`.

Let's take a look at one of them:

```
<soapenv:Envelope xmlns:soapenv="http://schemas.xmlsoap.org/soap/envelope/"
  xmlns:xsd="http://www.w3.org/2001/XMLSchema"
  xmlns:xsi="http://www.w3.org/2001/XMLSchema-instance">
<soapenv:Header>
  <wsse:Security xmlns:wsse="http://schemas.xmlsoap.org/ws/2002/04/secext"
    soapenv:actor="" soapenv:mustUnderstand="0">
    <xenc:ReferenceList xmlns:xenc="http://www.w3.org/2001/04/xmlenc#">
    <xenc:DataReference URI="EncryptedBody"></xenc:DataReference>
    </xenc:ReferenceList>
  </wsse:Security>
</soapenv:Header>
<soapenv:Body>
  <xenc:EncryptedData xmlns:xenc="http://www.w3.org/2001/04/xmlenc#"
```

```
  Id="EncryptedBody" Type="http://www.w3.org/2001/04/xmlenc#Element">
    <xenc:EncryptionMethod Algorithm="http://www.globus.org/2002/04/xmlenc#gssapi-
    </xenc:EncryptionMethod>
    <ds:KeyInfo xmlns:ds="http://www.w3.org/2000/09/xmldsig#">
      <ds:KeyName>00000000-0141-fab6-00000000-00000112d7ae</ds:KeyName>
    </ds:KeyInfo>
    <xenc:CipherData>
    <xenc:CipherValue>

    FwMAARBYgJbPOGPUfDe/rRt0Xig5Nvaf5xCUmLHMCUW3+kMsxgfA10ft319im/mn3PJGlLYoRo8M
    FfYMAgctOFjCPYEalpa1IhI+A28OmaVRUnCxw6QeynuQ6Op77D2p4frGd14tq0hB621JUcl3gY0F
    S4QuSZ/D5yjG+rO9mZUwL+5Y6K3umr1y8h65Cn4IQYd4uBuFYasaTOCX66PIhq+MkcfTimbHWkjG
    9o1cWcZJCgTiby4fhUsFjjzkTOFPHdY6lMMjC0qo4a5OJTq0ifjQfahjzCwfV/vsyMP7a21eAbZN
    G4ZHL0MCoIWQcZgRjFnSI28z3ZnjWTavqt3ByEbJDxGW+DnOVTikuVevLQdtZy86OA==
    </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. This is because GSI uses *message-level* security. This means that the message's format (SOAP) is not encrypted, but the contents of the message are (highlighted in bold). If we wanted the whole thing encrypted (both the message format and the message's content) we would need to use *transport-level* security. This type of security was supported by GSI in the past, but is now deprecated.

Chapter 14. The Security Configuration File

The example in the previous section used some pretty simple security: everything is encrypted, and there is no authorization decision being made. In this section we are going to see how we can start tweaking security in GT3. For example, we'll be able to specify that some methods have to be invoked securely while others can be invoked without any security at all. This is done by creating a custom *security configuration file*.

Writing a custom configuration file

Remember that, in the previous section, our example used a default security configuration file that simply said "everything has to be invoked securely". To do this, we included the following parameter in the deployment descriptor:

```
<service name="progtutorial/security/first/MathService" provider="Handler" style="

  <-- ... -->

  <parameter name="securityConfig"
    value="org/globus/ogsa/impl/security/descriptor/gsi-security-config.xml"/>

  <-- ... -->

<service>
```

This file (gsi-security-config.xml) is included with the toolkit. However, to gain more control over the security aspects of our service (instead of making *everything* secure), we'll have to write our own custom configuration file. In this file we'll be able to control to aspects of security on a per-method basis:

- **Authentication method**: We can specify what authentication method must be used by any client that wants to invoke the method. For example, we'll be able to specify that method FOO must be invoked with full encryption, while method BAR can be invoked simply with a digital signature (which guarantees integrity but not privacy). We can also specify that a method can be invoked with no security at all.
- **Runtime identity**: A service always runs under a certain identity. We can actually specify what identity the service must run under, although the practical use of this particular feature won't be apparent until we see delegation.

In this section we are going to write two configuration files: one to tweak the authentication methods and one to tweak the runtime identity. We will test both of them with *separate* clients to see how they react. We won't have to write any new service or GWSDL file, since we can use the ones from the previous section. We only need to add two new services to the WSDD file, each with different configuration files. In fact, if you take a look at the WSDD file `$TUTORIAL_DIR/org/globus/progtutorial/services/security/first/server-deploy.wsdd` you'll notice that those two services are already there:

```
<service name="progtutorial/security/first/MathAuthService" provider="Handler" sty

  <-- ... -->

  <parameter name="securityConfig"
    value="org/globus/progtutorial/services/security/first/config/security-config-

  <-- ... -->

<service>
```

```
<service name="progtutorial/security/first/MathRunAsService" provider="Handler" st

  <-- ... -->

  <parameter name="securityConfig"
    value="org/globus/progtutorial/services/security/first/config/security-config-

  <-- ... -->

<service>
```

In the next pages we'll learn how to write the security configuration files and give both services a try.

Setting authentication methods

We'll start by writing a configuration file that tweaks the authentication methods. First things first: What exactly is the security configuration file? It is a very simple XML file with the following root element:

```
<securityConfig xmlns="http://www.globus.org"
  xmlns:math="http://www.globus.org/namespaces/2004/02/progtutorial/MathService">

  <-- ... -->

<securityConfig>
```

Notice how we have to declare our service's namespace (as `math`). This is necessary since we are going to refer to the methods of our service (so the security configuration file must be aware of its namespace).

The <securityConfig> element can contain several <method> elements. Each of these <method> elements will allow us to configure the security options of an individual method. Right now, we are going to modify the authentication methods of all three methods in our MathService: `add`, `subtract`, and `getValue`.

```
<securityConfig xmlns="http://www.globus.org"
  xmlns:math="http://www.globus.org/namespaces/2004/02/progtutorial/MathService">
```

```
  <method name="math:add">
    <-- ... -->
  </method>

  <method name="math:subtract">
    <-- ... -->
  </method>

  <method name="math:getValue">
    <-- ... -->
  </method>

</securityConfig>
```

Notice how the method's name (in the `name` attribute) has the namespace prefix.

Next, each <method> element will contain a set of tags that allow us to configure that method. To modify the authentication method we'll need to include a <auth-method> element inside the <method> element. This element, in turn, can contain any of the following XML elements:

- <none>: When this element is included, the method can be invoked without any security.
- <pkey>: Use public-key authentication. In GT3.2, this kind of authentication is called GSI Secure Message.
- <gsi> Use GSI Secure Conversation.

All three can be used as empty elements (<none/>, <pkey/>, <gsi/>). However, as we'll see shortly, the <gsi> element can contain another element to further configure the GSI conversation.

In the following examples we will be using only <none> and <gsi> authentication. GSI Secure Message (<pkey>) also implies a secure communication, but is less feature-rich than <gsi> (GSI Secure Conversation). For example, it doesn't support encryption or delegation. However, it is faster and has less overhead, so the lack of features can be acceptable if performance is a big issue. The Stub security options appendix describes how to configure a client for GSI Secure Message.

No authentication

For now, let's start with the simplest case. The getValue method will have no security.

```
<method name="math:getValue">

  <auth-method>
    <none/>
  </auth-method>
</method>
```

GSI authentication

The <gsi> element can contain a <protection-level> element which can allow us to specify the protec-

tion level of the conversation: integrity and/or privacy. To specify which one we want, we have to include one (or both) of these empty elements inside the <protection-level> element:

- <integrity/>: The secure conversation must ensure integrity by including a digital signature. The message itself, however, will not be encrypted (so privacy is not ensured). This means the client stub *must* set Constants.GSI_SEC_CONV to Constants.SIGNATURE.
- <privacy/>: The secure conversation must ensure privacy by encrypting the message. This means that the client stub *must* set Constants.GSI_SEC_CONV to Constants.ENCRYPTION.

For example, let's suppose we want to make sure integrity is guaranteed in all invocations of the subtract method. The corresponding <method> element would look like this:

```
<method name="math:subtract">
  <auth-method>
    <gsi>
      <protection-level>
        <integrity/>
      </protection-level>
    </gsi>
  </auth-method>
</method>
```

However, this configuration forces subtract invocations to *only* use integrity protection. An invocation using encryption, for example, would fail. To allow both integrity and privacy, we can include *both* elements inside the <protection-level> element. For example, we will configure the add method that way:

```
<method name="math:add">
  <auth-method>
    <gsi>
      <protection-level>
        <integrity/>
        <privacy/>
      </protection-level>
    </gsi>
  </auth-method>
</method>
```

The whole file would look like this:

```
<securityConfig xmlns="http://www.globus.org"
  xmlns:math="http://www.globus.org/namespaces/2004/02/progtutorial/MathService">

<method name="math:add">
  <auth-method>
    <gsi>
      <protection-level>
        <integrity/>
        <privacy/>
      </protection-level>
    </gsi>
```

```
  </auth-method>
</method>

<method name="math:subtract">
  <auth-method>
    <gsi>
      <protection-level>
        <integrity/>
      </protection-level>
    </gsi>
  </auth-method>
</method>

<method name="math:getValue">
  <auth-method>
    <none/>
  </auth-method>
</method>

<!-- Default for other methods -->
<auth-method>
  <gsi/>
</auth-method>

</securityConfig>
```

Note

This file is `$TUTORIAL_DIR/org/globus/progtutorial/services/security/first/config/security-config-auth.xml`

Testing the different authentication methods

Compile and deploy

If you were expecting to perform that tedious compile + deploy routine *again*, don't worry: the two services described in this section were deployed along with the previous section's service. Remember that we're only changing some parameters in the deployment descriptor; we're reusing *all* the server-side code. However, if you haven't compiled and deployed the example from the previous chapter, you should do so right now to be able to try out the example client which is going to be described now.

Note

This is also a good moment to remind you that you should be using the downloadable examples included with the tutorial. What's that? You *still* haven't read the Introduction to the GT3 Security Services part of the tutorial? If you haven't done so yet, please do, and remember to use the downloadable examples available in the tutorial website [http://www.casa-sotomayor.net/gt3-tutorial] to follow this part of the tutorial.

The clients

We are going to invoke the methods on this instance with three different clients. This will allow us to observe how the server denies certain invocations if they don't meet the conditions specified in the security configuration file. These three clients are:

- **Encryption Client** : Configured to request an encrypted conversation. `$TUTORIAL_DIR/org/globus/progtutorial/clients/MathService/ClientGSIConvEncrypt.java`
- **Signed Client** : Configured to request a signed conversation (integrity). `$TUTORIAL_DIR/org/globus/progtutorial/clients/MathService/ClientGSIConvSigned.java`
- **No Security Client** : Configured to request a non secure conversation. `$TUTORIAL_DIR/org/globus/progtutorial/clients/MathService/ClientNoSecurity.java`

Remember we used the `ClientGSIConvEncrypt` client in the previous section. The other two clients (`ClientGSIConvSigned` and `ClientNoSecurity`) are exactly the same and only differ in the stub properties they set to configure security.

Encryption client

The encryption client sets the following stub properties:

```
((Stub)math)._setProperty(Constants.GSI_SEC_CONV,Constants.ENCRYPTION);
((Stub)math)._setProperty(Constants.AUTHORIZATION,NoAuthorization.getInstance());
```

Let's compile this client:

```
javac \
-classpath ./build/classes/:$CLASSPATH \
org/globus/progtutorial/clients/MathService/ClientGSIConvEncrypt.java
```

And now, let's run it:

```
java \
-classpath ./build/classes/:$CLASSPATH \
org/globus/progtutorial/clients/MathService/ClientGSIConvEncrypt \
http://127.0.0.1:8080/ogsa/services/progtutorial/security/first/MathAuthService \
5
```

You should see the following output:

```
Added 5

ERROR: GSI Secure Conversation (signature only) authentication required for
"{http://www.globus.org/namespaces/2004/02/progtutorial/MathService}subtract" oper

Current value: 5
```

Let's take a look at what just happened:

- We can invoke `add` because we're allowing both encrypted and signed conversations.
- We can't invoke `subtract` because we're only allowing signed conversations.
- No problem accessing `getValue` since no security is required. However, this doesn't mean we *can't* use security.

Signed Client

This client makes a secure invocation using only a digital signature which guarantees integrity but not privacy. To do this, we have to set the following stub properties:

```
((Stub)math)._setProperty(Constants.GSI_SEC_CONV,Constants.SIGNATURE);
((Stub)math)._setProperty(Constants.AUTHORIZATION,NoAuthorization.getInstance());
```

Let's compile and run the client:

```
javac \
-classpath ./build/classes/:$CLASSPATH \
org/globus/progtutorial/clients/MathService/ClientGSIConvSigned.java
```

```
java \
-classpath ./build/classes/:$CLASSPATH \
org/globus/progtutorial/clients/MathService/ClientGSIConvSigned \
http://127.0.0.1:8080/ogsa/services/progtutorial/security/first/MathAuthService \
5
```

You should see the following output:

```
Added 5

Subtracted 1

Current value: 9
```

What just happened?

- We can invoke `add` and `subtract` because, in both methods, we are allowing signed conversations.
- No problem accessing `getValue` since no security is required.

No security

Since this client has no security at all, we don't have to set any stub properties. Let's go straight to compiling and running:

```
javac \
-classpath ./build/classes/:$CLASSPATH \
org/globus/progtutorial/clients/MathService/ClientNoSecurity.java
```

```
java \
-classpath ./build/classes/:$CLASSPATH \
org/globus/progtutorial/clients/MathService/ClientNoSecurity \
http://127.0.0.1:8080/ogsa/services/progtutorial/security/first/MathAuthService \
5
```

You should see the following output:

```
ERROR: GSI Secure Conversation authentication required for
"{http://www.globus.org/namespaces/2004/02/progtutorial/MathService}add" operation

ERROR: GSI Secure Conversation (signature only) authentication required for
"{http://www.globus.org/namespaces/2004/02/progtutorial/MathService}subtract" oper

Current value: 9
```

The meaning of these messages are pretty straightforward: Both `add` and `subtract` fail since both require a secure conversation. However, we have no problem accessing `getValue` since no security is required.

Setting runtime identity

The second thing we can tweak in the security configuration file is the *runtime identity* of the service (in each method). This allows us to control the identity the service assumes during that invocation. Although the practical use of what we're about to see this might not be immediately apparent, it is specially relevant for credential delegation (which we will see soon).

First thing you should now is that, in a service invocation, there are three relevant subjects. Remember that a subject contains a *distinguished name* of the form CN=Borja Sotomayor, OU=GT3 Tutorial, O=Globus.

- **System subject**: This is the system's (the container's) subject. Unless we explicitly configure the container to use a different set of credentials, this subject will take its value from the subject of the user that's running the container. For example, if we use the globus account to run the container, the system subject will be O=Globus, OU=GT3 Tutorial, CN=Globus 3 Administrator
- **Service subject**: This is the subject of a particular service. A container can have services with different subjects. This subject is usually null, unless we specify credentials for the service or perform credential delegation (which we will see later on)
- **Invocation subject**: This subject depends on the runtime identity set in the security configuration file. If there is no security configuration file, this subject will be null.

Modifying the runtime identity through the security configuration file modifies the value of the invocation subject. We'll be able to give the invocation subject any of three possible identities:

- **The caller's identity**: This sets the invocation subject with the same value as the caller's subject (the client making the invocation)
- **The system's identity**: Sets the invocation subject with the system subject.
- **The service's identity**: Sets the invocation subject with the service subject (if the service doesn't have an identity, the system's identity will be used)

Setting this up is pretty straightforward. The <method> element can contain, besides an <auth-method> element, a <run-as> element. This element, in turn, can contain an empty <caller-identity/>, <system-identity/>, or <service-identity/>, which will determine the runtime identity of the method.

We are going to configure each of our three methods with a different runtime identity. The security configuration would look like this:

```
<securityConfig xmlns="http://www.globus.org"
  xmlns:math="http://www.globus.org/namespaces/2004/02/progtutorial/MathService">

<method name="math:add">
  <run-as>
    <caller-identity/>
  </run-as>
</method>

<method name="math:subtract">
  <run-as>
    <system-identity/>
  </run-as>
</method>

<method name="math:getValue">
  <run-as>
    <service-identity/>
  </run-as>
</method>

</securityConfig>
```

Note

This file is `$TUTORIAL_DIR/org/globus/progtutorial/services/security/first/config/security-config-runas.xml`

Testing the different runtime identities

Compile and deploy

Just like in the previous example, there is no need to compile and deploy, since this examples was also bundled with the first example we deployed.

The Client

We need to invoke all three methods and then look at the server-side logs to see what the system, service, and invocation subject are in each invocation. This is where the logSecurityInfo method we included in the service is going to come in handy. To invoke all three methods we can directly use the client from the previous section: `$TUTORIAL_DIR/org/globus/progtutorial/clients/MathService/ClientGSIConvEncrypt.java`

Let's run the client again

```
java \
-classpath ./build/classes/:$CLASSPATH \
org/globus/progtutorial/clients/MathService/ClientGSIConvEncrypt \
http://127.0.0.1:8080/ogsa/services/progtutorial/security/first/MathRunAsService \
5
```

Let's take a close look at all the server-side logs.

Logs from the add method (running as caller identity)

You should see the following when the `add` method is invoked:

```
INFO: SECURITY INFO FOR METHOD 'add'
INFO: The caller is:/O=Globus/OU=GT3 Tutorial/CN=Borja Sotomayor

INFO: INVOCATION SUBJECT
INFO: Subject:
        Principal: /O=Globus/OU=GT3 Tutorial/CN=Borja Sotomayor
        Private credential: org.globus.gsi.gssapi.GlobusGSSCredentialImpl@dea768

INFO: SERVICE SUBJECT
INFO: NULL

INFO: SYSTEM SUBJECT
INFO: Subject:
        Principal: /O=Globus/OU=GT3 Tutorial/CN=Globus 3 Administrator
        Private credential: org.globus.gsi.gssapi.GlobusGSSCredentialImpl@2d0483
```

Notice how the invocation subject assumes the identity of the caller.

`subtract` method (running as system identity)

You should see the following when the `subtract` method is invoked:

```
INFO: SECURITY INFO FOR METHOD 'subtract'
INFO: The caller is:/O=Globus/OU=GT3 Tutorial/CN=Borja Sotomayor

INFO: INVOCATION SUBJECT
INFO: Subject:
        Principal: /O=Globus/OU=GT3 Tutorial/CN=Globus 3 Administrator
        Private credential: org.globus.gsi.gssapi.GlobusGSSCredentialImpl@2d0483

INFO: SERVICE SUBJECT
INFO: NULL

INFO: SYSTEM SUBJECT
INFO: Subject:
```

```
        Principal: /O=Globus/OU=GT3 Tutorial/CN=Globus 3 Administrator
        Private credential: org.globus.gsi.gssapi.GlobusGSSCredentialImpl@2d0483
```

The invocation subject assumes the identity of the system. Since the container is being run by the `globus` account, the invocation subject is equal to the `globus` user's subject.

`getValue` method (running as service identity)

You should see the following when the `getValue` method is invoked:

```
INFO: SECURITY INFO FOR METHOD 'getValue'
INFO: The caller is:/O=Globus/OU=GT3 Tutorial/CN=Borja Sotomayor

INFO: INVOCATION SUBJECT
INFO: Subject:
        Principal: /O=Globus/OU=GT3 Tutorial/CN=Globus 3 Administrator
        Private credential: org.globus.gsi.gssapi.GlobusGSSCredentialImpl@2d0483

INFO: SERVICE SUBJECT
INFO: NULL

INFO: SYSTEM SUBJECT
INFO: Subject:
        Principal: /O=Globus/OU=GT3 Tutorial/CN=Globus 3 Administrator
        Private credential: org.globus.gsi.gssapi.GlobusGSSCredentialImpl@2d0483
```

The invocation subject should assume the identity of the service. However, since the service has no identity (NULL), it assumes the identity of the system (the `globus` account)

Chapter 15. Access Control with Gridmaps

Gridmaps are one of the forms of server-side GSI authorization. They allow us to control access to our grid services, and also play an important roles in higher level services. A *gridmap* is basically an ACL (Access Control List) that allows us to specify what users have access to a service.

Adding gridmap authorization is easy. We just have to add two lines to the WSDD (plus create the gridmap file). Although we could directly reuse most of the code of the previous two sections, we're going to start working in a new directory (`$TUTORIAL_DIR/org/globus/progtutorial/services/security/gridmap/`) since this example will also be used in the next section (credential delegation).

The gridmap file

First of all, we need to create the *gridmap file*. This file has a list of distinguished names that are allowed access to a service. The file also maps each distinguished name to a user account. However, we won't be using that feature now, since it is used by higher-level services which the tutorial currently doesn't cover.

The format of the file is very simple. One line for each user which is allowed access. Each line has two fields separated by whitespace: the distinguished name and the user account. Since the distinguished name usually contains whitespace, it is placed between "quotation marks".

For example, a gridmap file which gave my distinguished name access to a service could be the following:

```
"/O=Globus/OU=GT3 Tutorial/CN=Borja Sotomayor" borja
```

Note

Save this file as `$GLOBUS_LOCATION/gridmapfile`.

Configuring gridmap authorization

To configure a service to use gridmap authorization, we need to modify the authorization parameters in the WSDD file, which will now have the following value: `gridmap`. We also need to add a new parameter called `gridmap` which points to the gridmap file. Our WSDD file could look something like this:

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

  <service name="tutorial/security/gridmap/MathFactoryService" provider="Handler"

    <!-- ... -->


    <parameter name="authorization" value="gridmap"/>
    <parameter name="gridmap" value="gridmapfile"/>
```

```
    <!-- ... -->
  </service>
</deployment>
```

Note

This file is `$TUTORIAL_DIR/org/globus/progtutorial/services/security/gridmap/server-deploy.wsdd`

Since paths in the WSDD file are relative to `$GLOBUS_LOCATION` we can simply write gridmapfile as the value of the gridmap parameter to refer to `$GLOBUS_LOCATION/gridmapfile`. Should the gridmap file be outside the `$GLOBUS_LOCATION` directory tree, then you should include a complete (absolute) path.

The grid service

Service interface

The service interface is the same as the one in the previous two sections (add, subtract, and getValue).

Service implementation

The code is very similar to the one from the previous two sections. The only real difference is that we're producing less logging information (We'll be using this service in the next section, and the only 'interesting' info we need is the caller's identity). The new implementation can be found in `$TUTORIAL_DIR/org/globus/progtutorial/services/security/gridmap/impl/MathProvider.java`

Compile and deploy

We'll need to compile this new service:

```
./tutorial_build.sh \
org/globus/progtutorial/services/security/gridmap \
schema/progtutorial/MathService/Math.gwsdl
```

Now, let's deploy it (remember to do this from the `globus` account)

```
ant deploy \
-Dgar.name=$TUTORIAL_DIR/build/lib/org_globus_progtutorial_services_security_gridm
```

Since we'll be producing some logs on the server side, add this line to the end of `$GLOBUS_LOCATION/ogsilogging.properties`:

```
org.globus.progtutorial.services.security.first.impl.MathProvider=console,info
```

Starting the container

Before starting the container (with the `globus` account) make sure you've created a proxy certificate. Once you've done that, start the container as usual:

```
globus-start-container
```

Testing the gridmap

We'll give the service a try with the same client used in the previous chapters: `$TUTORIAL_DIR/org/globus/progtutorial/clients/MathService/ClientGSIConvEncrypt.java`

If you haven't already done so, compile the client:

```
javac \
-classpath ./build/classes/:$CLASSPATH \
org/globus/progtutorial/clients/MathService/ClientGSIConvEncrypt.java
```

Now, run the client from your user account (in my case, the `borja` account):

```
java \
-classpath ./build/classes/:$CLASSPATH \
org/globus/progtutorial/clients/MathService/ClientGSIConvEncrypt \
http://127.0.0.1:8080/ogsa/services/progtutorial/security/gridmap/MathService \
5
```

You should get a pretty normal output:

```
Added 5
Current value: 5
```

However, try to run it from any other account (for example, the `globus` account). You will get this nasty little error:

```
org.globus.ogsa.impl.security.authorization.AuthorizationException:
Gridmap authorization failed:
peer "/O=Globus/OU=GT3 Tutorial/CN=Globus 3 Administrator" not in gridmap file.
```

The error message speaks for itself: Since the user running the client is not in the gridmap file, it has been denied access to the server.

Chapter 16. Delegation

In this section we are going to see two practical examples of credential delegation. Before seeing these examples, this might be a good time to reread the page on credential delegation.

The first example is going to be a very simple example based on the very first secure example we saw (in directory `$TUTORIAL_DIR/org/globus/progtutorial/services/security/first/`). The modified files can be found in `$TUTORIAL_DIR/org/globus/progtutorial/services/security/delegation_first/`. This example will allow us to see the basic mechanism that activates delegation and, using the server-side logs, we will verify that delegation is working properly.

However, this first example doesn't allow us to see the full potential of delegation. This is why, after that example, we will write a more complex example based on the illustration we saw when explaining credential delegation. The second example will include *two* services. The client will delegate its credentials on the first service, and that service will use those delegated credentials to invoke the second service. We will be able to see that, when delegation is not activated, the example doesn't work (because the second service expects the client's credentials, not the first service's credentials). Then, once we activate delegation, we will see how everything works perfectly.

A first approach at delegation

As mentioned before, this first example is based on the first secure service we saw. This means we'll only see the differences between both examples. However, remember that the full code for this example can be found in `$TUTORIAL_DIR/org/globus/progtutorial/services/security/delegation_first/`.

Activating delegation on the client side

The first thing we have to do is modify the client to tell it to delegate its credentials to the service. This is done simply by setting the following stub property:

```
((Stub)math)._setProperty(GSIConstants.GSI_MODE,GSIConstants.GSI_MODE_FULL_DELEG);
```

Note

The client can be found in `$TUTORIAL_DIR/org/globus/progtutorial/clients/MathService/ClientDelegation.java`

Activating delegation on the server side

But for delegation to be complete, we need to do two things on the server side:

- The particular method we want to invoke must be configured to run with the caller's identity. In other words, the *invocation subject* must be set to the caller's identity. We saw how to do this earlier, in the runtime identity part of the security configuration file chapter.

- We have to tell the service to assume the identity of the caller. Remember from the runtime identity page that service subject was always NULL, unless we delegated credentials on the service. We will be able to do this by adding one simple line of code.

Setting the runtime identity

To set the runtime identity, we will be able to reuse the security configuration file we used in the runtime identity example. Remember that, in that example, the add method was configured to run under the caller's identity. The security configuration file for this example can be found in `$TUTORIAL_DIR/org/globus/progtutorial/services/security/delegation_first/config/security-config-runas.xml`

Setting the service owner

To make the service assume the invocation subject as its subject, we have to add the following line in each method where we want to perform delegation:

```
SecurityManager.getManager().setServiceOwnerFromContext(base);
```

For example, in the add method:

```
public void add(int a) throws RemoteException
, SecurityException
{

  SecurityManager.getManager().setServiceOwnerFromContext(base);
  logSecurityInfo("add");
  value = value + a;
}
```

Compile and deploy

Now, let's build the service:

```
./tutorial_build.sh \
org/globus/progtutorial/services/security/delegation_first \
schema/progtutorial/MathService/Math.gwsdl
```

And deploy it (from the `globus` account):

```
ant deploy \
-Dgar.name=$TUTORIAL_DIR/build/lib/org_globus_progtutorial_services_security_deleg
```

Finally, before you restart the container, add the following line to the `$GLOBUS_LOCATION/ogsilogging.properties` file:

```
org.globus.progtutorial.services.security.delegation_first.impl.MathProvider=conso
```

Compiling and running the client

Let's compile the client:

```
javac \
-classpath ./build/classes/:$CLASSPATH \
org/globus/progtutorial/clients/MathService/ClientDelegation.java
```

Finally, we run the client:

```
java \
-classpath ./build/classes/:$CLASSPATH \
org/globus/progtutorial/clients/MathService/ClientDelegation \
http://127.0.0.1:8080/ogsa/services/progtutorial/security/delegation/MathService \
5
```

The output on the client side should be pretty normal. We need to take a close look at the server side logs to verify that delegation is, in fact, working. Look at the add method (which runs under the caller's identity):

```
INFO: SECURITY INFO FOR METHOD 'add'
INFO: The caller is:/O=Globus/OU=GT3 Tutorial/CN=Borja Sotomayor

INFO: INVOCATION SUBJECT
INFO: Subject:
        Principal: /O=Globus/OU=GT3 Tutorial/CN=Borja Sotomayor
        Private credential: org.globus.gsi.gssapi.GlobusGSSCredentialImpl@f4ca49

INFO: SERVICE SUBJECT
INFO: Subject:
        Principal: /O=Globus/OU=GT3 Tutorial/CN=Borja Sotomayor
        Private credential: org.globus.gsi.gssapi.GlobusGSSCredentialImpl@f4ca49

INFO: SYSTEM SUBJECT
INFO: Subject:
        Principal: /O=Globus/OU=GT3 Tutorial/CN=Globus 3 Administrator
        Private credential: org.globus.gsi.gssapi.GlobusGSSCredentialImpl@1f88fbd
```

Notice how the service subject is not only no longer NULL...it's the caller's identity! Holy identity theft, Batman! :-)

As for the subtract and getValue methods, the service subject is also no longer NULL. However, since they're being run under the system and subject identity (respectively), we see the globus account subject in the service subject.

```
INFO: SECURITY INFO FOR METHOD 'subtract'
INFO: The caller is:/O=Globus/OU=GT3 Tutorial/CN=Borja Sotomayor
INFO: INVOCATION SUBJECT
INFO: Subject:
```

```
        Principal: /O=Globus/OU=GT3 Tutorial/CN=Globus 3 Administrator
        Private credential: org.globus.gsi.gssapi.GlobusGSSCredentialImpl@1f88fbd

INFO: SERVICE SUBJECT
INFO: Subject:
        Principal: /O=Globus/OU=GT3 Tutorial/CN=Globus 3 Administrator
        Private credential: org.globus.gsi.gssapi.GlobusGSSCredentialImpl@1f88fbd

INFO: SYSTEM SUBJECT
INFO: Subject:
        Principal: /O=Globus/OU=GT3 Tutorial/CN=Globus 3 Administrator
        Private credential: org.globus.gsi.gssapi.GlobusGSSCredentialImpl@1f88fbd
```

So, we've seen that delegation actually *does* work. However, this example isn't exactly what you could call 'exciting'. However, I promise the next example is guaranteed to positively thrill you!

Description of this example

We are now going to start working on a more elaborate delegation example which uses *two* services:

- **PhysicsService**: This is a new service we will program from scratch.
- **MathService**: This is, in fact, the MathService with gridmap authorization. We won't need to make a single modification to this service.

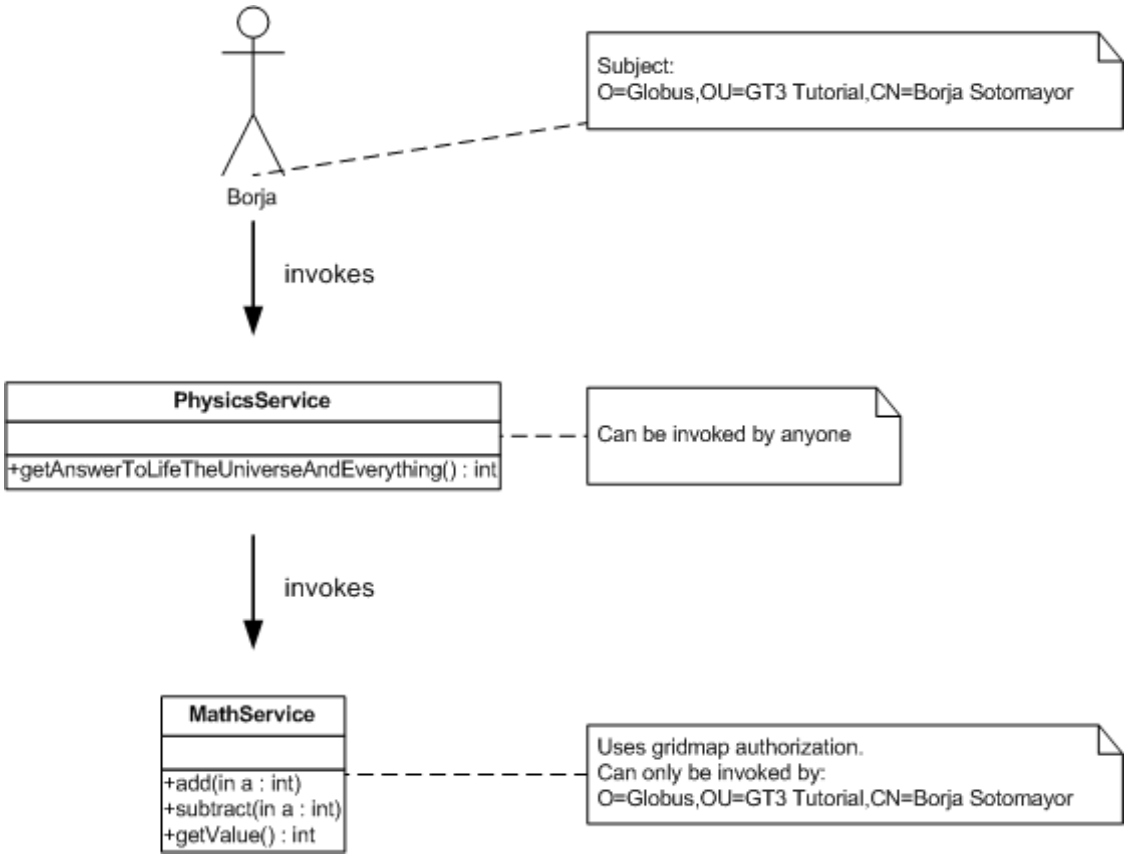
In our example, PhysicsService will be responsible for undertaking an extremely complex calculation. For this, it will need to invoke MathService several times. However, there's a catch:

- We will be invoking the PhysicsService from our user account. In my case, this account has a certificate with subject `O=Globus,OU=GT3 Tutorial,CN=Borja Sotomayor`
- PhysicsService allows any user to access its methods. Therefore, the client application will have no trouble accessing PhysicsService.
- MathService, on the other hand, only allows one user to access its methods. We'll be using the same service and gridmap as seen in the gridmap section, so this should be your user account (unless you've modified the gridmap file). In my case, my gridmap file only allows access to the user with subject `O=Globus,OU=GT3 Tutorial,CN=Borja Sotomayor`

This all means that, for the example to work, PhysicsService *must* invoke MathService using the `O=Globus,OU=GT3 Tutorial,CN=Borja Sotomayor` credential. Since PhysicsService is running in a container that has a different set of credentials (those of the `globus` user), PhysicsService will only be able to access MathService if the client application (run by the `borja` user account) delegates its credentials.

In fact, to show that this is all true, we will test PhysicsService without delegation and with delegation.

The following is an illustration summarizing the whole example:



As shown in the illustration, PhysicsService has a single method `getAnswerToLifeTheUniverseAndEverything` which will be invoked by the client application. This method, in turn, will invoke the `add` method in MathService several times.

PhysicsService

We'll start by seeing the service and client that don't perform delegation. Later on, we'll see how using this same code, and adding just a couple of lines of code, delegation can be activated. All the code for the non-delegating example can be found here: $TUTORIAL_DIR/org/globus/progtutorial/services/security/delegation/

Service interface

The service interface of the PhysicsService is very simple, as it only has a single method. It can be found here: `$TUTORIAL_DIR/schema/progtutorial/PhysicsService/Physics.gwsdl`

Service implementation

The implementation is going to be pretty long, since this is the first time we have a service that invokes another service. You can find the code here: `$TUTORIAL_DIR/org/globus/progtutorial/services/security/delegation/impl/PhysicsProviderNoDelegation.java`

However, since this code is lengthy, we're going to take the time to take a look at it step-by-step. First of all, instead of fitting all the code into the `getAnswerToLifeTheUniverseAndEverything`

method, we're going to divide everything into several private methods. The 'skeleton' of our implementation looks something like this:

```
package org.globus.progtutorial.services.security.delegation.impl;

// import statements

public class PhysicsProviderNoDelegation implements OperationProvider
{
  // Create this class's logger
  static Log logger = LogFactory.getLog(PhysicsProvider.class.getName());

  // Operation provider properties
  private static final String namespace =
    "http://www.globus.org/namespaces/2004/02/progtutorial/PhysicsService";

  private static final QName[] operations =
    new QName[]
      {new QName(namespace, "getAnswerToLifeTheUniverseAndEverything")};


  static final String mathFactoryURL =
  "http://127.0.0.1:8080/ogsa/services/progtutorial/security/gridmap/MathService";


  public int getAnswerToLifeTheUniverseAndEverything() throws RemoteException, Sec
  {
  }


  private void makeStubSecure(Object stub)
  {
  }

  private MathPortType getReferenceToMathService() throws RemoteException
  {
  }

  private void logSecurityInfo()
  {
  }
}
```

Let's take a look at what all these attributes and methods do.

mathFactoryURL attribute

```
static final String mathServiceGSH =
"http://127.0.0.1:8080/ogsa/services/progtutorial/security/gridmap/MathService";
```

Yes, this is a very big no-no. In the real world, a GSH should never be hard-coded into your service code, but obtained from an index service. However, we're using a hard-coded GSH for simplicity.

getAnswerToLifeTheUniverseAndEverything method

This is the only public remote method, and the one which will invoke the `add` method in MathService. Therefore, a lot of its code will be dedicated to communicating with MathService.

Before we start doing anything, we'll need a couple of variables:

```
MathPortType math;
int answer;
```

First of all, we're going to call the `logSecurityInfo` (which outputs the same data as in the previous examples) so we can check if delegation is working or not:

```
logSecurityInfo();
```

Next, we get a reference to the MathPortType.

```
math = getReferenceToMathService();
```

Now, we invoke the `add` method in MathService a couple times, thus obtaining The Answer.

```
for(int i=0; i<7; i++)
{
  logger.info("Invoking 'add' method...");
  math.add(6);
  logger.info("Invoked 'add' method");
}
```

Next, we call MathService's `getValue` to get the value of The Answer...

```
logger.info("Invoking 'getValue' method...");
answer = math.getValue();
logger.info("Invoked 'getValue' method");
```

Finally, we return The Answer:

```
return answer;
```

The whole thing would look something like this:

```
public int getAnswerToLifeTheUniverseAndEverything() throws RemoteException, Secur
{
  MathPortType math;
  int answer;

  logSecurityInfo();

  math = getReferenceToMathService();

  // Find the answer to life, the universe, and everything.
  // This is accomplished by invoking the 'add' method in the MathService!
  for(int i=0; i<7; i++)
  {
    logger.info("Invoking 'add' method...");
    math.add(6);
    logger.info("Invoked 'add' method");
  }

  logger.info("Invoking 'getValue' method...");
  answer = math.getValue();
  logger.info("Invoked 'getValue' method");

  return answer;
}
```

logSecurityInfo method

The logSecurityInfo method is very similar to the one we used in previous examples, with a couple minor modifications. We'll use it to write out the caller's identity, and the invocation, subject, and service subject.

Other private methods

The rest of the private methods perform operations which we've seen in many of the client applications (getting a reference to an instance, making a stub secure, etc.)

The getReferenceToMathInstance obtains a MathPortType stub from the locator.

```
private MathPortType getReferenceToMathService() throws RemoteException
{
  URL GSH = null;
  try{
   GSH = new java.net.URL(mathServiceGSH);
  } catch(Exception e){ }

  logger.info("Obtaining reference to MathService...");
  MathServiceGridLocator mathLocator =   new MathServiceGridLocator();
  MathPortType math = mathLocator.getMathServicePort(GSH);
  makeStubSecure(math);
  logger.info("Obtained reference to MathService");
  return math;
}
```

Notice getReferenceToMathService calls method makeStubSecure to set the stub's security properties.

```
private void makeStubSecure(Object stub)
{
  ((Stub)stub)._setProperty(Constants.GSI_SEC_CONV,Constants.ENCRYPTION);
  ((Stub)stub)._setProperty(Constants.AUTHORIZATION,NoAuthorization.getInstance())
}
```

Compiling and deploying

Deployment descriptor

The WSDD file is pretty straightforward, and similar to the one used in the security configuration file examples. The file can be found here: `$TUTORIAL_DIR/org/globus/progtutorial/services/security/delegation/server-deploy.wsdd`. This file also includes the deployment descriptor for the next example (the PhysicsService with delegation activated)

However, we'll need to create a custom security configuration file to specify that the service's method must be run under the caller's identity. Otherwise, delegation will not take place. The security configuration file, found at `$TUTORIAL_DIR/org/globus/progtutorial/services/security/delegation/config/security-config.xml`, looks like this:

```
<securityConfig xmlns="http://www.globus.org"
  xmlns:physics="http://www.globus.org/namespaces/2004/02/progtutorial/PhysicsServ

<method name="physics:getAnswerToLifeTheUniverseAndEverything">

  <run-as>
    <caller-identity/>
  </run-as>
  <auth-method>
    <gsi/>
  </auth-method>
</method>

<auth-method>
  <gsi/>
</auth-method>

</securityConfig>
```

Compile and deploy

Before we compile and deploy, make sure you add this line to the `$GLOBUS_LOCATION/ogsilogging.properties` file:

```
org.globus.progtutorial.services.security.delegation.impl.PhysicsNoDelegationProvi
```

Build the service:

```
./tutorial_build.sh \
org/globus/progtutorial/services/security/delegation \
schema/progtutorial/PhysicsService/Physics.gwsdl
```

And deploy the GAR file (from the `globus` account):

```
ant deploy \
-Dgar.name=$TUTORIAL_DIR/build/lib/org_globus_progtutorial_services_security_deleg
```

A non-delegating client

Remember that the service still isn't completely ready to perform delegation. We're going to use a client that doesn't perform delegation, to see how MathService denies access to PhysicsService because it isn't using the adequate credentials.

The client itself is pretty simple. We'll just invoke the `getAnswerToLifeTheUniverseAndEverything` in PhysicsService. The client's only parameter is the service's GSH.

```
package org.globus.progtutorial.clients.PhysicsService;

import org.globus.progtutorial.stubs.PhysicsService.service.PhysicsServiceGridLoca
import org.globus.progtutorial.stubs.PhysicsService.PhysicsPortType;

import org.globus.ogsa.impl.security.Constants;
import org.globus.ogsa.impl.security.authorization.NoAuthorization;

import java.net.URL;
import javax.xml.rpc.Stub;

public class ClientNoDelegation
{
  public static void main(String[] args)
  {
    try
    {
      // Get command-line arguments
      URL GSH = new java.net.URL(args[0]);

      // Get a reference to the MathService instance
      PhysicsServiceGridLocator physicsLocator =   new PhysicsServiceGridLocator()
      PhysicsPortType physics = physicsLocator.getPhysicsServicePort(GSH);

      // Setup security options
      ((Stub)physics)._setProperty(Constants.GSI_SEC_CONV,Constants.ENCRYPTION);
      ((Stub)physics)._setProperty(Constants.AUTHORIZATION,NoAuthorization.getInst

      // Call remote method 'add'
      int answer = physics.getAnswerToLifeTheUniverseAndEverything();

      System.out.println("Answer: " + answer);
```

```
    }catch(Exception e)
    {
      System.out.println("ERROR:" + e.getMessage());
    }
  }
}
```

Note

This file is `$TUTORIAL_DIR/org/globus/progtutorial/clients/PhysicsService/ClientNoDelegation.java`

Now, let's compile the client:

```
javac \
-classpath ./build/classes/:$CLASSPATH \
org/globus/progtutorial/clients/PhysicsService/ClientNoDelegation.java
```

And run the client:

```
java \
-classpath ./build/classes/:$CLASSPATH \
org/globus/progtutorial/clients/PhysicsService/ClientNoDelegation \
http://127.0.0.1:8080/ogsa/services/progtutorial/security/delegation/PhysicsServic
```

You should get this nasty little error:

```
org.globus.ogsa.impl.security.authorization.AuthorizationException:
Gridmap authorization failed:
peer "/O=Globus/OU=GT3 Tutorial/CN=Globus 3 Administrator" not in gridmap file.
```

A closer look at the server logs reveals the following:

```
INFO: -------- BEGIN SECURITY INFO --------
INFO: Caller: /O=Globus/OU=GT3 Tutorial/CN=Borja Sotomayor
INFO: Invocation subject:/O=Globus/OU=GT3 Tutorial/CN=Borja Sotomayor
INFO: Service subject:
NULL
INFO: System subject:/O=Globus/OU=GT3 Tutorial/CN=Globus 3 Administrator
INFO: --------  END SECURITY INFO  --------
```

Even though we're running under the caller's identity (the invocation subject is correctly set to the caller's subject), the *service subject* is still NULL. Since this is subject is NULL, the container will use the service subject (`/O=Globus/OU=GT3 Tutorial/CN=Globus 3 Administrator`) to invoke MathService. However, that subject isn't in MathService's gridmap, and that's why we get a "Gridmap authorization failed" error message.

So...let's add delegation to the PhysicsService and see how it all finally works out.

Adding delegation

Adding delegation in the client

We need to make the client delegate its credentials. As we saw in the first delegation example, we can accomplish this by setting a stub security option:

```
((Stub)math)._setProperty(GSIConstants.GSI_MODE,GSIConstants.GSI_MODE_FULL_DELEG);
```

Accepting delegation on the server side

To make the service use the delegated credential as its own, we'll need to add the following line in the `getAnswerToLifeTheUniverseAndEverything` method, right before calling `logSecurityInfo`:

```
SecurityManager.getManager().setServiceOwnerFromContext(base);
```

The modified version of PhysicsService (with delegation activated) can be found in `$TUTORIAL_DIR/org/globus/progtutorial/services/security/delegation/impl/PhysicsProvider.java`. As noted in the previous page, this new service was deployed along with the previous example (the non-delegating PhysicsService)

Compiling, deploying, and running the client

Compiling, deploying, and running the client can be done following the instructions in the previous page. Just make sure you work with the `$TUTORIAL_DIR/org/globus/progtutorial/services/security/delegation` directory, *not* in the `$TUTORIAL_DIR/org/globus/progtutorial/services/security/delegation_notdelegating` directory.

```
javac \
-classpath ./build/classes/:$CLASSPATH \
org/globus/progtutorial/clients/PhysicsService/ClientDelegation.java
```

```
java \
-classpath ./build/classes/:$CLASSPATH \
org/globus/progtutorial/clients/PhysicsService/ClientDelegation \
http://127.0.0.1:8080/ogsa/services/progtutorial/security/delegation/PhysicsServic
```

Once you run the client, you should see the following output:

```
Answer: 42
```

Note

Yes, this *is* the Answer to Life, the Universe, and Everything. Further details in *The Hitchhiker's Guide to the Galaxy* :-)

If we take a look at the server logs, we'll see how delegation is really taking place. First, let's look at the security info written by the PhysicsService:

```
INFO: -------- BEGIN SECURITY INFO --------
INFO: Caller: /O=Globus/OU=GT3 Tutorial/CN=Borja Sotomayor
INFO: Invocation subject:
/O=Globus/OU=GT3 Tutorial/CN=Borja Sotomayor
INFO: Service subject:
/O=Globus/OU=GT3 Tutorial/CN=Borja Sotomayor
INFO: System subject:/O=Globus/OU=GT3 Tutorial/CN=Globus 3 Administrator
INFO: --------  END SECURITY INFO  --------
```

Notice how, this time, both the invocation subject *and* the service subject are the same as the caller's subject.

Now let's take a look at the messages output by MathService:

```
INFO: Creating MathService instance...
INFO: Created MathService instance
INFO: Obtaining reference to MathService instance...
INFO: Obtained reference to MathService instance
INFO: Invoking 'add' method...
INFO: 'add' invoked by: /O=Globus/OU=GT3 Tutorial/CN=Borja Sotomayor
INFO: Invoked 'add' method
INFO: Invoking 'add' method...
INFO: 'add' invoked by: /O=Globus/OU=GT3 Tutorial/CN=Borja Sotomayor
INFO: Invoked 'add' method
INFO: Invoking 'add' method...
INFO: 'add' invoked by: /O=Globus/OU=GT3 Tutorial/CN=Borja Sotomayor
INFO: Invoked 'add' method
INFO: Invoking 'add' method...
INFO: 'add' invoked by: /O=Globus/OU=GT3 Tutorial/CN=Borja Sotomayor
INFO: Invoked 'add' method
INFO: Invoking 'add' method...
INFO: 'add' invoked by: /O=Globus/OU=GT3 Tutorial/CN=Borja Sotomayor
INFO: Invoked 'add' method
INFO: Invoking 'add' method...
INFO: 'add' invoked by: /O=Globus/OU=GT3 Tutorial/CN=Borja Sotomayor
INFO: Invoked 'add' method
INFO: Invoking 'add' method...
INFO: 'add' invoked by: /O=Globus/OU=GT3 Tutorial/CN=Borja Sotomayor
INFO: Invoked 'add' method
INFO: Invoking 'getValue' method...
INFO: 'getValue' invoked by: /O=Globus/OU=GT3 Tutorial/CN=Borja Sotomayor
INFO: Invoked 'getValue' method
INFO: Destroying MathService instance...
INFO: Destroyed MathService instance
```

Even though the MathService is being invoked by PhysicsService (which is being run by the `globus` account), it is using the caller's delegated credentials when making the call to the `add` method.

Pretty nifty, huh?

Part IV. Appendices

Table of Contents

Appendix A. How to...

...write a GWSDL description of your Grid Service

This "How to..." appendix shows how to write a simple GWSDL description of a PortType in a step-by-step fashion. Although it should be easy to extrapolate from the example we are going to see and create other simple GWSDL files, this is not meant as an exhaustive guide of GWSDL or WSDL. 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.

That said, let's start writing GWSDL! We are going to write the GWSDL description corresponding to the following Java interface:

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

  public void subtract(int a);

  public int getValue();
}
```

This is the interface of many of the examples of the tutorial.

First of all, we have to write the root element of the GWSDL file, which is <definitions>.

```
<?xml version="1.0" encoding="UTF-8"?>
<definitions name="MathService"
  targetNamespace="http://www.globus.org/namespaces/2004/02/progtutorial/MathServi
  xmlns:tns="http://www.globus.org/namespaces/2004/02/progtutorial/MathService"
  xmlns:ogsi="http://www.gridforum.org/namespaces/2003/03/OGSI"
  xmlns:gwsdl="http://www.gridforum.org/namespaces/2003/03/gridWSDLExtensions"
  xmlns:xsd="http://www.w3.org/2001/XMLSchema"
  xmlns="http://schemas.xmlsoap.org/wsdl/">

</definitions>
```

This tag has two important attributes:

- **name** : The 'name' of the GWSDL file. Not related with the name of the PortType.
- **targetNamespace** : The target namespace of the GWSDL file. This means that all the PortTypes and operations defined in this GWSDL file will belong to this namespace. In case you're not familiar with XML namespace, they're basically a way of grouping similar 'things' into a group. I'm using the somewhat vague term 'things' because XML Namespace is used not only in WSDL/GWSDL, 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 that URL, 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 an OGSI GWSDL file with definitions we'll need later on.

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

Now we're going to define our PortType, using the <gwsdl:portType> tag (if this were ordinary WSDL, we would simply use a <portType> tag). Notice how we declared the gwsdl namespace in the root element.

```
<definitions ... ">

<gwsdl:portType name="MathPortType" extends="ogsi:GridService">
  <operation name="add">
    <input message="tns:AddInputMessage"/>
    <output message="tns:AddOutputMessage"/>
    <fault name="Fault" message="ogsi:FaultMessage"/>
  </operation>
  <operation name="subtract">
    <input message="tns:SubtractInputMessage"/>
    <output message="tns:SubtractOutputMessage"/>
    <fault name="Fault" message="ogsi:FaultMessage"/>
  </operation>
  <operation name="getValue">
    <input message="tns:GetValueInputMessage"/>
    <output message="tns:GetValueOutputMessage"/>
    <fault name="Fault" message="ogsi:FaultMessage"/>
  </operation>
</gwsdl:portType>

</definitions>
```

The <gwsdl:portType> tag has two important attributes:

- **name**: Name of the PortType.
- **extends**: As mentioned in the first GT3 Core chapter, this is one of the main differences with plain WSDL. These attributes allow us to define our PortType as an extension of an existing PortType. In this case, we extend from ogsi:GridService PortType, which all Grid Services must extend from.

Inside the <gwsdl:portType> we have an <operation> tag for each method in our PortType: add, subtract, and getValue. They are all very similar, so let's take a closer look at the add <operation> tag:

```
<operation name="add">
  <input message="tns:AddInputMessage"/>
  <output message="tns:AddOutputMessage"/>
  <fault name="Fault" message="ogsi:FaultMessage"/>
</operation>
```

The <operation> tag has an <input> tag, an <output> tag and a <fault> tag. These three tags have a message attribute, which specifies what message should be passed along when the operation is invoked (input message), when it returns successfully (output message) or when an error occurs (fault message). The fault message is defined in the OGSI GWSDL we included earlier. However, we'll need to define the messages of our operations. The following are the messages for the add operation. The messages for the subtract operation are identical (just change 'add' for 'subtract' :-)

```
<?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>

&lt!-- 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 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:

```
<types>
<xsd:schema targetNamespace="http://www.globus.org/namespaces/2004/02/progtutorial
  attributeFormDefault="qualified"
  elementFormDefault="qualified"
  xmlns="http://www.w3.org/2001/XMLSchema">


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

  <xsd:element name="addResponse">
```

```
    <xsd:complexType/>
  </xsd:element>

</xsd:schema>
</types>

&lt!-- Messages -->

&lt!-- 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 GWSDL document.

The add element (which, remember, is part of the input message of the add operation) contains an element called value which is the parameter of the add operation (notice how the type attribute is equal to xsd:int, the integer type in XML Schema).

On the other hand, the addResponse element (part of the output message of the add operation, i.e. the return value) contains no elements at all, since the add operation doesn't return anything.

The rest of the <messages> and element types are defined in a similar fashion. The whole GWSDL file would be:

```
<?xml version="1.0" encoding="UTF-8"?>
<definitions name="MathService"
  targetNamespace="http://www.globus.org/namespaces/2004/02/progtutorial/MathServi
  xmlns:tns="http://www.globus.org/namespaces/2004/02/progtutorial/MathService"
  xmlns:ogsi="http://www.gridforum.org/namespaces/2003/03/OGSI"
  xmlns:gwsdl="http://www.gridforum.org/namespaces/2003/03/gridWSDLExtensions"
  xmlns:xsd="http://www.w3.org/2001/XMLSchema"
  xmlns="http://schemas.xmlsoap.org/wsdl/">

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

<types>
<xsd:schema targetNamespace="http://www.globus.org/namespaces/2004/02/progtutorial
  attributeFormDefault="qualified"
  elementFormDefault="qualified"
  xmlns="http://www.w3.org/2001/XMLSchema">
  <xsd:element name="add">
    <xsd:complexType>
      <xsd:sequence>
        <xsd:element name="value" type="xsd:int"/>
      </xsd:sequence>
    </xsd:complexType>
  </xsd:element>
  <xsd:element name="addResponse">
    <xsd:complexType/>
  </xsd:element>
  <xsd:element name="subtract">
    <xsd:complexType>
      <xsd:sequence>
        <xsd:element name="value" type="xsd:int"/>
      </xsd:sequence>
    </xsd:complexType>
```

```
  </xsd:element>
  <xsd:element name="subtractResponse">
    <xsd:complexType/>
  </xsd:element>
  <xsd:element name="getValue">
    <xsd:complexType/>
  </xsd:element>
  <xsd:element name="getValueResponse">
    <xsd:complexType>
      <xsd:sequence>
        <xsd:element name="value" type="xsd:int"/>
      </xsd:sequence>
    </xsd:complexType>
  </xsd:element>
</xsd:schema>
</types>

<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="GetValueInputMessage">
  <part name="parameters" element="tns:getValue"/>
</message>
<message name="GetValueOutputMessage">
  <part name="parameters" element="tns:getValueResponse"/>
</message>

<gwsdl:portType name="MathPortType" extends="ogsi:GridService">
  <operation name="add">
    <input message="tns:AddInputMessage"/>
    <output message="tns:AddOutputMessage"/>
    <fault name="Fault" message="ogsi:FaultMessage"/>
  </operation>
  <operation name="subtract">
    <input message="tns:SubtractInputMessage"/>
    <output message="tns:SubtractOutputMessage"/>
    <fault name="Fault" message="ogsi:FaultMessage"/>
  </operation>
  <operation name="getValue">
    <input message="tns:GetValueInputMessage"/>
    <output message="tns:GetValueOutputMessage"/>
    <fault name="Fault" message="ogsi:FaultMessage"/>
  </operation>
</gwsdl:portType>

</definitions>
```

Summing up, the basic steps involved in writing a GWSDL file would be the following:

1. Write the root element <definitions>

2. Write the <gwsdl:PortType>

3. Write an input and output <message> for each operation in the PortType.

4. Write the <types>

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 GWSDL. 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.

...setup the GT3 command line clients

Most of the examples in the tutorial depend on a set of very handy command line clients included in GT3. For example, to create a Grid Service instance we can use the ogsi-create-service command:

```
ogsi-create-service
  http://localhost:8080/ogsa/services/tutorial/core/factory/MathFactoryService
```

Without the command line clients, we would need to run a Java class, which is not as easy to remember as ogsi-create-service.

```
java org.globus.ogsa.client.CreateService
     http://localhost:8080/ogsa/services/tutorial/core/factory/MathFactoryService
```

Furthermore, using the Java class directly, we would first have to set all the necessary environment variables first (the CLASSPATH, etc.). The command line clients take care of all this automatically.

However, these command line clients are, by default, not setup. To do so, follow these simple steps:

1. Set an environment variable called `GLOBUS_LOCATION` with the path of the root of your GT3 installation. For example, `/usr/local/gt3/`

2. Run the command `ant setup` from the root of your GT3 installation. Make sure you run this with a user with write permissions on that directory.

3. All the command line clients will be created in `$GLOBUS_LOCATION/bin`. You can now run the commands from that directory.

4. (Optional) To run the commands from any directory, add `$GLOBUS_LOCATION/bin` to your `PATH` variable.

You can find more detailed instructions on how to setup the clients, along with a description of all the available clients, in the User's Guide included with GT3, which you can find in: $GLOBUS_LOCATION/docs/users_guide.html (11 - Command Line Clients)

Appendix B. Stub security options

This appendix describes the stub properties which can be modified to configure the security options on the client side.

GSI Secure Conversation

To use GSI Secure Conversation, with encryption:

```
stub._setProperty(Constants.GSI_SEC_CONV,Constants.ENCRYPTION);
```

To use GSI Secure Conversation, with integrity:

```
stub._setProperty(Constants.GSI_SEC_CONV,Constants.SIGNATURE);
```

GSI Secure Message

To use public-key authentication (<pkey> in the security configuration file), also known as GSI Secure Message:

```
stub._setProperty(Constants.GSI_XML_SIGNATURE,Boolean.TRUE);
```

To explicitly deactivate it:

```
stub._setProperty(Constants.GSI_XML_SIGNATURE,Boolean.FALSE);
```

Authorization

The GSI authorization types on the client side are described here [../security/gsi/authorization.html].

To use no authorization:

```
stub._setProperty(Constants.AUTHORIZATION,NoAuthorization.getInstance());
```

To use "self" authorization:

```
stub._setProperty(Constants.AUTHORIZATION,SelfAuthorization.getInstance());
```

To use host authorization:

```
stub._setProperty(Constants.AUTHORIZATION,HostAuthorization.getInstance());
```

Delegation

To perform no credential delegation at all:

```
stub._setProperty(GSIConstants.GSI_MODE,GSIConstants.GSI_MODE_NO_DELEG);
```

To perform full credential delegation:

```
stub._setProperty(GSIConstants.GSI_MODE,GSIConstants.GSI_MODE_FULL_DELEG);
```

To perform limited credential delegation:

```
stub._setProperty(GSIConstants.GSI_MODE,GSIConstants.GSI_MODE_LIMITED_DELEG);
```

Appendix C. Tutorial directory structure

The tutorial follows a very specific directory structure which clearly separates the interface files (GWSDL), 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, GWSDL, service, and client files) are located. The details of each type of file can be found below.

```
$TUTORIAL_DIR
  |
  |-- schema/
  |     |
  |     |-- progtutorial/          ----->  GWSDL files
  |
  |-- org/
        |
        |-- globus/
              |
              |-- progtutorial/
                    |
                    |-- services/   ----->  Service implementation files
                    |
                    |-- clients/    ----->  Client implementation files
```

Build files

All the files needed to build the examples are included in the root of `$TUTORIAL_DIR`:

- Ant build file
- Build script
- Namespace mappings file

GWSDL files

The `$TUTORIAL_DIR/schema/progtutorial/` directory contains one subdirectory for each different service interface described in the tutorial. These subdirectories contain the GWSDL 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.progtutorial.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 "GT3 Core" part of the tutorial, the implementation classes are placed inside this package:

```
org.globus.progtutorial.services.core.first
```

This will be the *base package* for this example. In general, the base package will have the following format:

```
org.globus.progtutorial.services.<part>.<example>
```

For example, the base package for the "delegation" example of the "GT3 Security Services" part is org.globus.progtutorial.service.**security.delegation** The directory corresponding to that base package would be:

```
$TUTORIAL_DIR/org/globus/progtutorial/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 $TUTORIAL_DIR/org/globus/progtutorial/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 $TUTORIAL_DIR/schema/progtutorial/**MathService**/ can be found in $TUTORIAL_DIR/org/globus/progtutorial/clients/**MathService**/

Appendix D. Frequently Asked Questions

Scope of this FAQ: This document is not intended as a general GT3 FAQ, but only aims to answer questions related to the tutorial. If you're looking for a complete GT3 FAQ, you can find one here [http://www.globus.org/toolkit/gt3-faq.html] (in the Globus website).

Q:
I get a NoClassDefFoundError when trying to run a GT3 command.

A:
Whenever you run a GT3 command without Ant (for example, to create a service instance, or when running the clients), you need to manually set a couple of environment variables first.

In GT3.0.x this is done by placing yourself in $GLOBUS_LOCATION and running the following command:

```
source ./setenv.sh
```

GT3.2.x users should do the following:

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

Remember, if you're using Ant, Ant takes care of setting the environment variables for you. If you still get that error, make sure GT3 and Ant are properly installed (double-check the environment variables).

Q:
I get a lot of cannot resolve symbol errors when trying to compile a client (mostly in Globus classes)

A:
The same answer to the previous question applies to this one. Since we're not using Ant to compile the clients, some environment variables must be set up first (the setenv.sh and globus-devel-env.sh scripts takes care of that). Also, you need to run this command *once* (not each time you want to compile a client).

If you get cannot resolve symbol errors on stub classes (MathPortType, MathServiceGridLocator, etc.) make sure you're using the `-classpath ./build/classes:$CLASSPATH` argument when running the Java compiler.

Q:
Can I use the 'handy multipurpose' buildfile for personal projects? If so, what limitations does it have?

A:
Feel free to use the handy buildfile and script (used throughout the tutorial) in your projects. However, take into account that they have certain limitations, since they assume a very specific directory structure. As long as any examples you write are similar to the ones shown in the tutorial, you should be safe. In the near future I hope to thoroughly document and expand the 'handy multipurpose' buildfile so it can be easily used in non-tutorial examples.