# The Globus Toolkit 3 Programmer's Tutorial


**Version 0.2.2**. See Change Log. See "To do" list.

**ATTENTION:** A new 'work-in-progress' version of the tutorial has been released at the following URL: http://www.casa-sotomayor.net/gt3-tutorial-working/. This new version includes sections on GT3 Security Services. However, please remember it is a *work in progress* (please read the warning at the top of the new version).

## Getting Started



## GT3 Core

- **Service Data in OGSA**
- **A service with Service Data**
- **A client that accesses Service Data**
- **The GridService Service Data**

- Notifications

  - **What are notifications?**
  - **A 'pull' notification service**
  - **A 'pull' notification client**
  - **A 'push' notification service**
  - **A 'push' notification client**

## How to...

- ...write a GWSDL description of your Grid Service.
- ...setup the GT3 command line clients (`globus-start-container`, `ogsi-create-service`, ...).

# Tutorial FAQ

*Borja Sotomayor* *28-Jul-2003*

# The Globus Toolkit 3 Programmer's Tutorial

## Change Log

**<-- Back**

### 0.2.2

28/07/2003

- Added two new FAQ entries, plus a link to one of them in the notifications section.
- Several minor bug and typo fixes.

### 0.2.1

30/06/2003

- Several minor bug and typo fixes.
- Commands for compiling the clients now correctly include the `-classpath ./build/classes:$CLASSPATH` parameter.
- Added new entry in FAQ related to compiling clients.

### 0.2

25/06/2003

- First version of the tutorial compliant with OGSI v1.0 Draft 29 (GT3 Alpha 4, Beta, and Final release).
- New sections:
    - Operation Providers
    - Logging
    - Callbacks
    - Lifecycle Management
    - "How to..."

### 0.1.x

March + April + May 2003

- Alpha-compliant version of the tutorial.

# The Globus Toolkit 3 Programmer's Tutorial

## "To do" list

**<-- Back**

This is a list of well defined tasks that have to be done in the tutorial, or that have been suggested by readers. I hope to eventually do them myself, but if anyone wants to volunteer to do one of them, or to help out with one, their assistance will be very appreciated :-) Please contact me if you'd like to help out.

- Installation: Write a step-by-step installation guide, instead of linking to other sites.
- Include instructions on how to deploy Grid Services in other host environments, such as Tomcat.
- Include instructions on how to use the handy multipurpose buildfile for other projects.
- Include a 'compileClient' task in the buildfile.
- GT3 Core (Lifecycle): Expand this section to include properties, persistent properties, `requestTerminationAfter` and `requestTerminationBefore`.
- GT3 Core (Service Data): Make introduction to Service Data a bit clearer. All the different concepts (Service Data Sets, Service Data Elements, type of the SDE, value of the SDE, multivalued SDEs, single valued SDEs, etc.) can be a bit confusing some times.
- GT3 Core (Service Data): Write an example that shows how a Service Data Set can have more than one Service Data Element.
- Add a section on the BottomUp and TopDown tools (replace handy multipurpose buildfile with these tools?)
- Instead of having one GWSDL file for each example, have two or three 'central' GWDSL files (since most of the interfaces in the examples are practically identical)

The following are more general objectives which following versions of the tutorial will (hopefully) cover:

- Include sections on GT3 higher-level services (Security Services, Base Services, Data Services)
- Migrate the whole tutorial from plain XHTML to a more manageable system (DocBook?). This should improve navegation, broken links, etc.
- Include a complete list of easy and simple examples, besides the ones seen in the tutorial.
- Include more "How to..." pages.
- Add JUnit modules for automated testing.
- The tutorial is primarily UNIX-oriented. It should also take into account Windows users.

**<-- Back**

*Borja Sotomayor*

# The Globus Toolkit 3 Programmer's Tutorial

## Introduction

**^Up^ Next -->**

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:

- **Introduction**: Includes information about the whole tutorial, as well as 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.)

**^Up^ Next -->**

*Borja Sotomayor*

# The Globus Toolkit 3 Programmer's Tutorial

## Introduction

### GT3 Prerequisite Documents

**<-- Previous ^Up^ Next -->**

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 (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 also be interested in taking a look at the 'Publications' section in the Globus website, 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. I. Foster, C. Kesselman, S. Tuecke.
- The Physiology of the Grid: An Open Grid Services Architecture for Distributed Systems Integration. I. Foster, C. Kesselman, J. Nick, S. Tuecke.

**<-- Previous ^Up^ Next -->**

*Borja Sotomayor*

# The Globus Toolkit 3 Programmer's Tutorial

## Introduction

### Audience

**<-- Previous ^Up^ Next -->**

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.

**<-- Previous ^Up^ Next -->**

*Borja Sotomayor*

# The Globus Toolkit 3 Programmer's Tutorial

## Introduction

### Assumptions

**<-- Previous ^Up^ Next -->**

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.

**<-- Previous ^Up^ Next -->**

*Borja Sotomayor*

# The Globus Toolkit 3 Programmer's Tutorial

## Introduction

### Related Documents

**<-- Previous ^Up^ Next -->**

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.

GT3 users have also contributed installation and programming guides. Although some of these versions still refer to Alpha and Beta versions of GT3, some users might find them useful.

- GT3 Alpha Installation (for Redhat 7.x): Complete installation guide. Includes information on configuring certificates, databases, and MMJFS. Written by Xin Ling.
- Step-By-Step Example for the Globus Toolkit 3.0: Written by Javier Cano.
- Grid Install for Windows 2000 Platform: Excellent starting point for Windows users. Includes plenty of screenshots. Written by Michael Schneider.

Once you've become a Grid Services expert, you might have to occasionally take a look at the Grid Services Specification (OGSI).

**<-- Previous ^Up^ Next -->**

*Borja Sotomayor*

# The Globus Toolkit 3 Programmer's Tutorial

## Introduction

### Document Conventions

**<-- Previous ^Up^ Next -->**

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");
  }
}
```

This is a comment related to the block of code.

### Shell commands

```
javac HelloWorld.java
```

### Author notes

> **NOTE**: This is a note by the author, referring to the document itself. For example, to point out that a section will be rewritten, that an example needs to be changed, etc. Usually these comments are left for the "To do" list, but sometimes it's interesting to point them out explicitly in the text.

**<-- Previous ^Up^ Next -->**

*Borja Sotomayor*

# The Globus Toolkit 3 Programmer's Tutorial

## Introduction

### Download tutorial files

**<-- Previous ^Up^ Next -->**

> These links are for the 0.2 files. For the 0.3 files please use the link in the index of the tutorial.

The following files are available for download:

- Tutorial examples. All the source files for the examples. The directory structure is the same as the one described in the tutorial.
- Handy multipurpose buildfile. This is the buildfile described in the Our handy multipurpose buildfile section.
- Handy multipurpose shell script. This is the script described in the Our handy multipurpose buildfile section.

**<-- Previous ^Up^ Next -->**

*Borja Sotomayor*

# The Globus Toolkit 3 Programmer's Tutorial

## Introduction

### About the author & acknowledgements

**<-- Previous ^Up^ Next -->**

The Globus Toolkit 3 Programmer's Tutorial is maintained by Borja Sotomayor, a Computer Science Engineer in the University of Deusto (Bilbao, Spain). You can find out more about me in my weblog, BorjaNet.

### Ackowledgements

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
- Leon Kuntz
- Jesús Marco
- All the Globus gurus who have reviewed the tutorial: Sam Meder, Thomas Sandholm, Von Welch.

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:

> Thomas Becker, Javier Cano, Qin Feng, Fernando Fraticelli, Steve Mock, Michael Schneider, Shiva Shankar Chetan, Nelson Sproul
>
> If you've reported a bug, typo, or helped out in any way, and you are not listed here, please do let me know!

**<-- Previous ^Up^ Next -->**

*Borja Sotomayor*

# The Globus Toolkit 3 Programmer's Tutorial

## Key Concepts

**<-- Previous ^Up^ Next -->**

There are certain key concepts that must be well understood before being able to program with GT3. This section 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.

**<-- Previous ^Up^ Next -->**

*Borja Sotomayor*

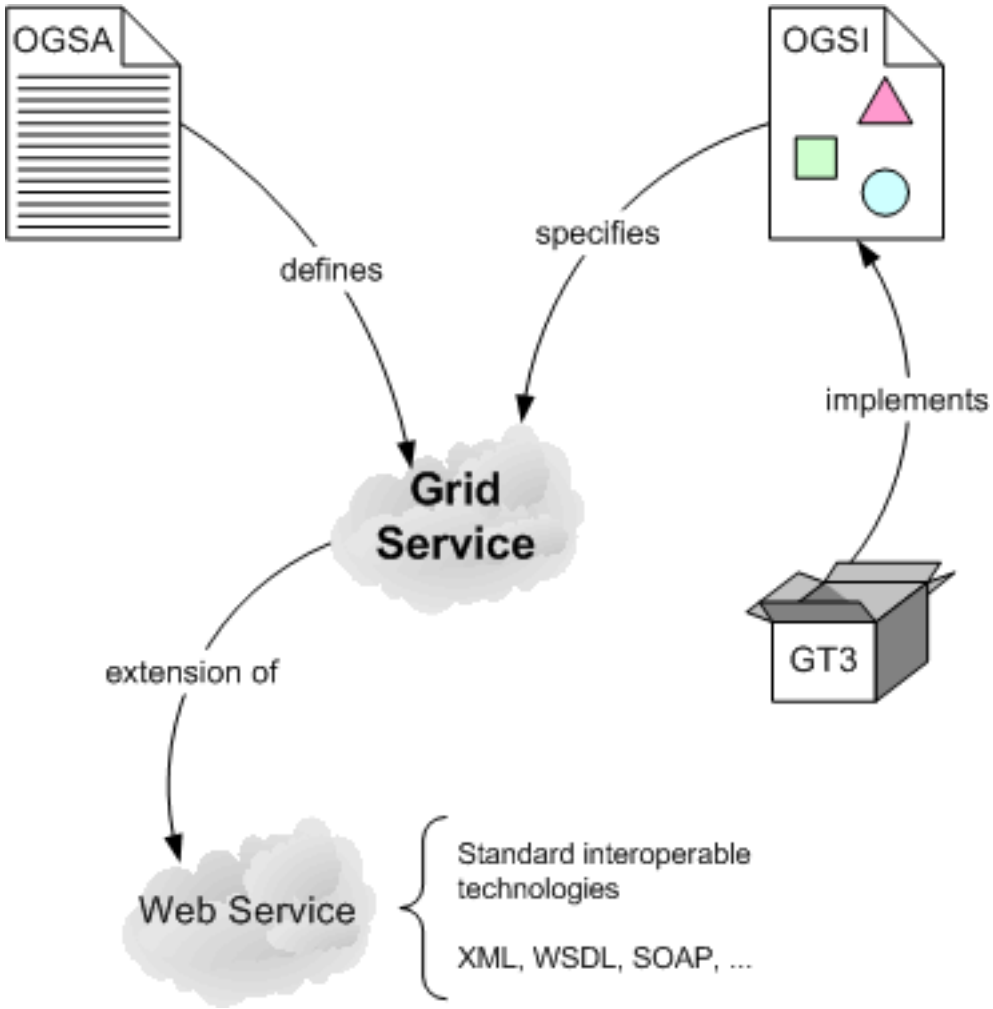
# The Globus Toolkit 3 Programmer's Tutorial

## Key Concepts

### OGSA, OGSI, and GT3

**<-- Previous ^Up^ Next -->**

The Globus Toolkit is a software toolkit that allows us to program grid-based applications. The third and latest version of the 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 Service world:




**Grid Services are defined by OGSA.** The Open Grid Services Architecture (OGSA) aims to define a new common and standard architecture for grid-based applications. RIght at the center of this new architecture is the concept of a Grid Service. OGSA *defines* what Grid Services are, what they should be capable of, what types of technologies they should be based on, but doesn't give a technical and detailed specification (which would be needed to implement a Grid Service). A link to the document

"Physiology of the Grid" (which introduces OGSA) can be found in the Prerequisite Documents section.

**Grid Services are specified by OGSI.** The Open Grid Services Infrastructure is a formal and technical specification of the concepts described in OGSA, including Grid Services. The Related Documents section has a link to the Grid Service Specification. This document is much more technical than the OGSA document.

**The Globus Toolkit 3 is an implementation of OGSI.** GT3 is a usable implementation of everything that is specified in OGSI (and, therefore, of everything that is defined in OGSA).

**Grid Services are based on Web Services.** Grid Services are an *extension* of Web Services. We'll see what Web Services are in the next page, and what Grid Services are in the page after that.

**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 all the plans, 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 make detailed blueprints that specify construction details (like where to put the master beams, the power cables, the plumbing, etc.). The engineer then passes all those blueprints to qualified professional workers (construction workers, electricians, plumbers, etc) who will actually build the house. We could say that OGSA (the definition) is the architect, OGSI (the specification) is the engineer, and GT3 (the implementation) is the workers.

**<-- Previous ^Up^ Next -->**

*Borja Sotomayor*

# The Globus Toolkit 3 Programmer's Tutorial

## Key Concepts

### A short introduction to Web Services

**<-- Previous ^Up^ Next -->**

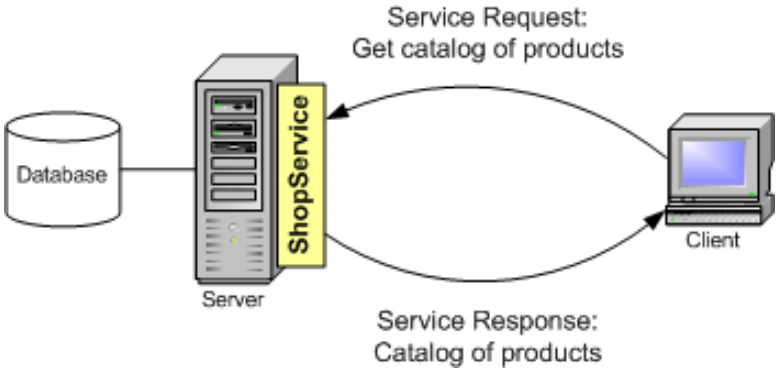
Web Services are the basis for Grid Services, which are the cornerstones of OGSA and OGSI (and, therefore, of the Globus Toolkit 3). Understanding the Web Services architecture is fundamental to using GT3 and programming Grid Services.

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:

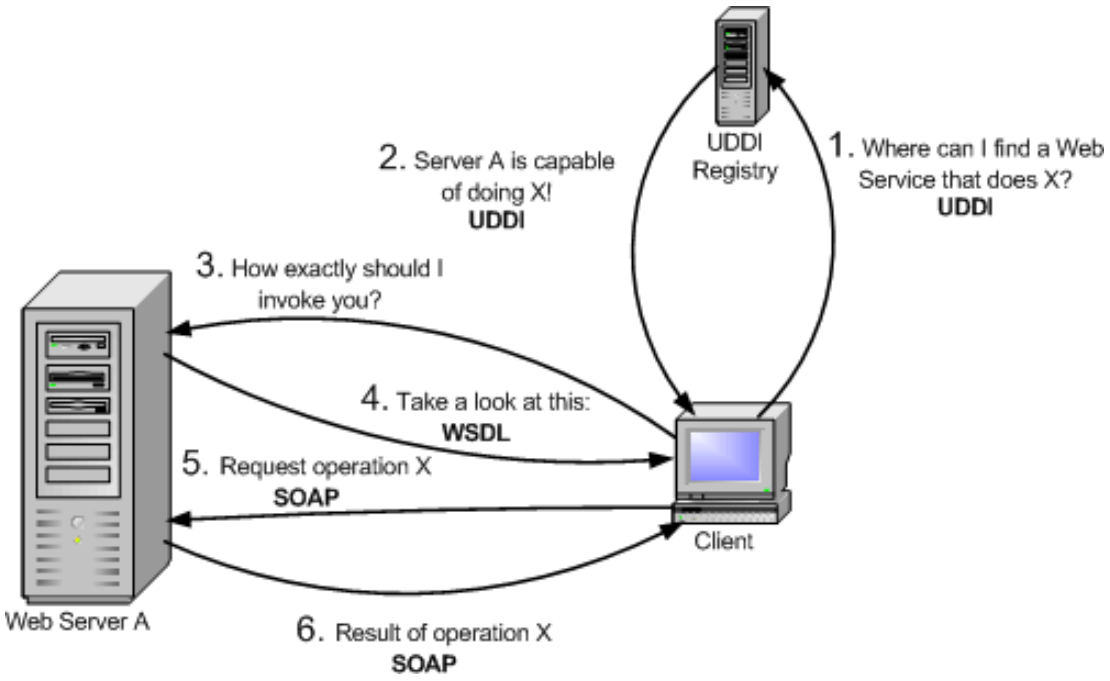
- Overhead. Transmitting all your data in XML is obviously not as efficient as using a proprietary binary code. What you win in portability, you lose in efficiency. Even so, this overhead is usually acceptable for most applications, but you will

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 oriented toward *highly coupled* distributed systems, where the client and the server are very dependent on each other, Web Services are oriented towards *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 invokation. 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.

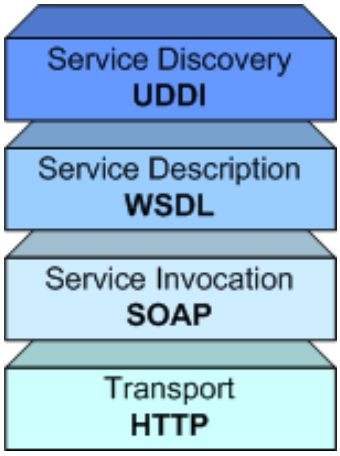
For example, the UDDI registry might have replied with the following URI:

```
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 uninteligible 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 Integration). 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 respondes. 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.

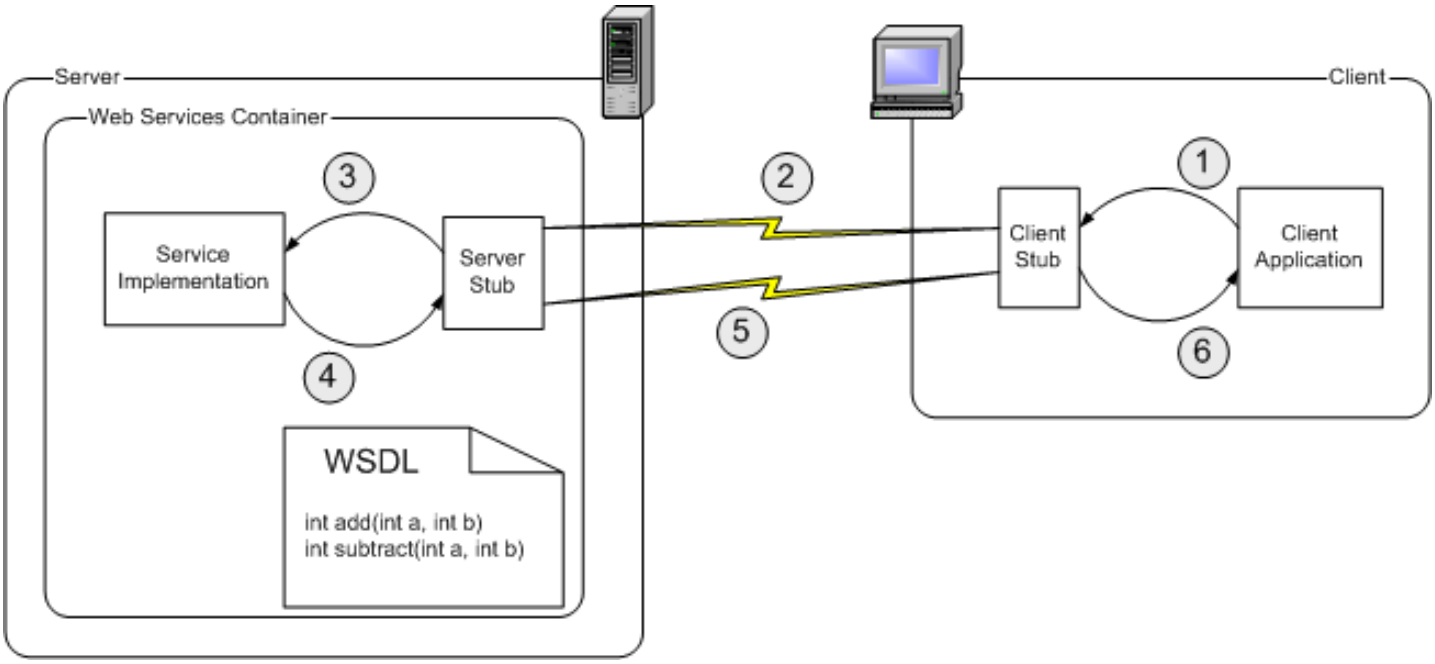
First of all, you should know that despite having a lot of protocols and languages floating around, Web Services programmers usually never write a single line of SOAP or WSDL. Once we've reached a point where our client application needs to invoke a Web Service, we *delegate* that task on a piece of 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 used) which will be in charge of interpreting requests and *fowarding* 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 *marshalling* 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 *unmarshalling*)
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, the same organization responsible for XML, HTML, CSS, etc.

**<-- Previous ^Up^ Next -->**

*Borja Sotomayor*

# The Globus Toolkit 3 Programmer's Tutorial

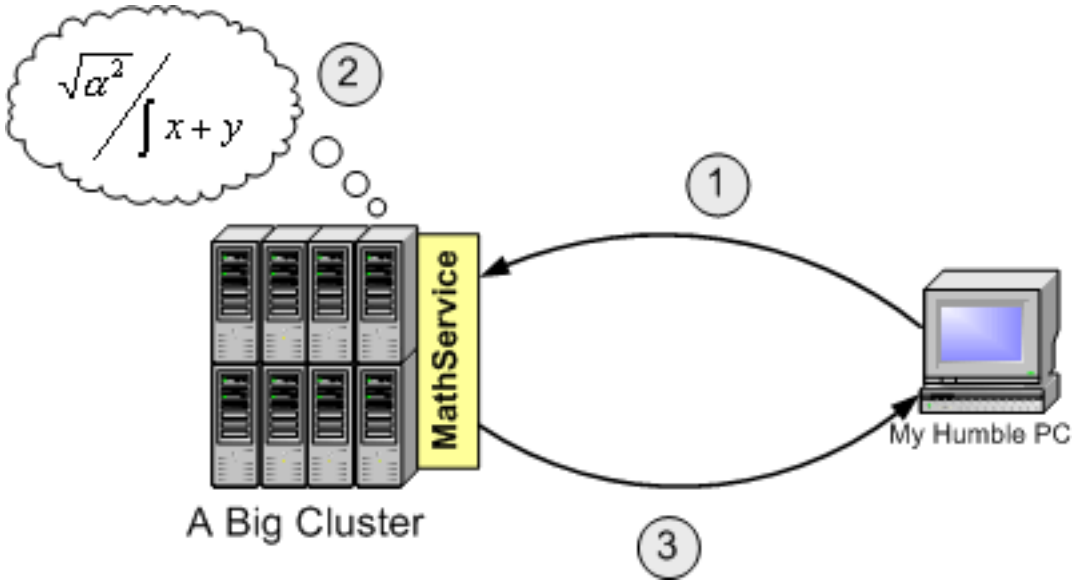
## Key Concepts

### What is a Grid Service?

**<-- Previous ^Up^ Next -->**

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 making 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.

Let's take a tour of these improvements with 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!



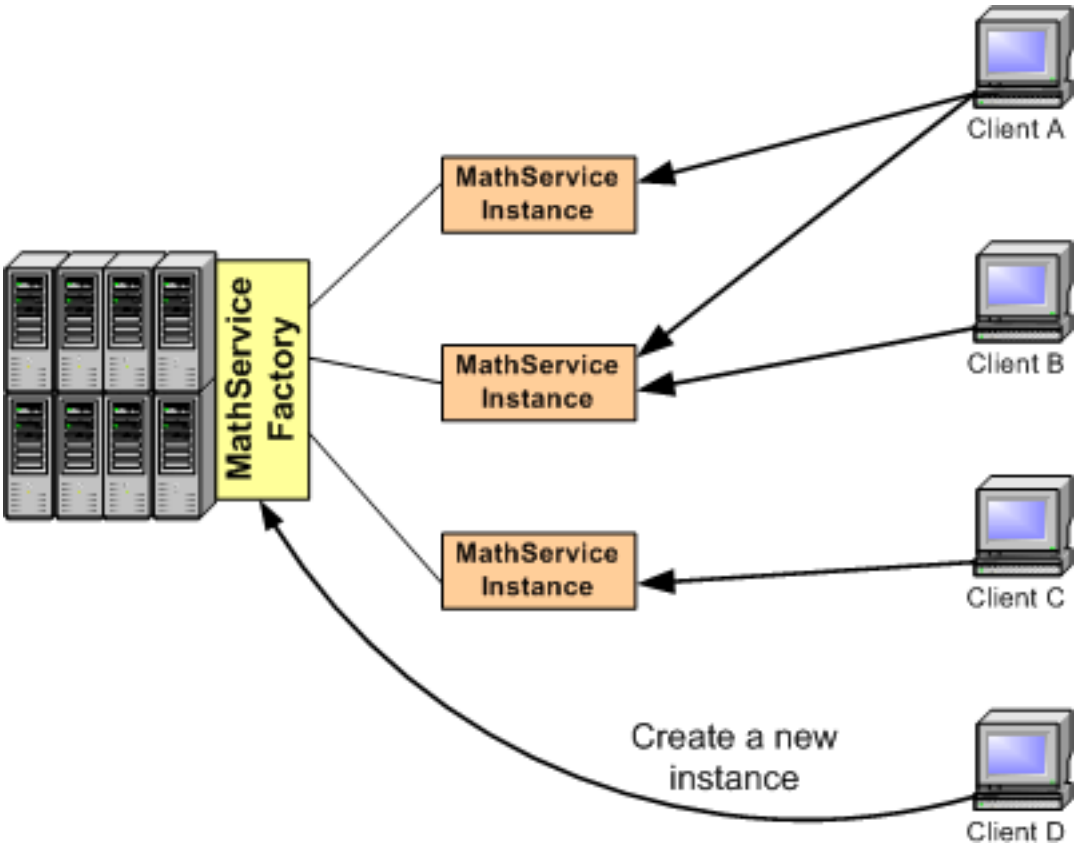

We could implement a Math Web Service called *MathService* which offered operations such as `SolveReallyBigSystem()`, `SolveFermatsLastTheorem()`, etc. At first, we would be able to perform typical Web Service invocations:

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* and *non-transient*. "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. 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. Certainly, this isn't a very elegant solution!

## Factories

Grid Services solve both problems by proposing a *factory* approach to Web Services. Instead of having one big stateless MathService shared by all users, we actually have a central MathService factory which is 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 (the instance will eventually be destroyed). The lifetime of an instance can vary from application to application. Usually, we 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.

## Other Grid Services Improvements

Factories are, by far, the most interesting improvement offered by Grid Services. However, Grid Services have more to offer:

- **Two implementation approaches**: A Grid Service can be implemented either by inheriting from a *skeleton class* or by using a delegation model, where incoming calls are *delegated* to a series of classes called *operation providers*.
- **Lifecycle management**: Grid Services provide the necessary tools, such as callback functions during special moments in a Grid Service's lifetime (creation time, destruction time, etc.), to effectively manage its lifecycle (for example, to make Grid Services persistent).
- **Service Data**: A Grid Service can have a set of associated service data that describes it. This is not to be confused with WSDL, which describes details like methods, protocols, etc. Service data is particularly useful to index Grid Services according to their characteristics and capabilities.
- **Notifications**: We can configure a Grid Service 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.

## GSH & GSR

Finally, before moving on, we need to learn two very important acronyms which we will see a lot from now on. They are related to Grid Services *addressing*. In the previous section we saw that Web Services are addressed with URIs. Since Grid Service *are* Web Services, they are also addressed with URIs. However, a "Grid Service URI" is called the *Grid Service Handle*, or simply GSH.

Each GSH must be unique. There cannot be two Grid Services (or Grid Service instances) 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.

**<-- Previous ^Up^ Next -->**

*Borja Sotomayor*

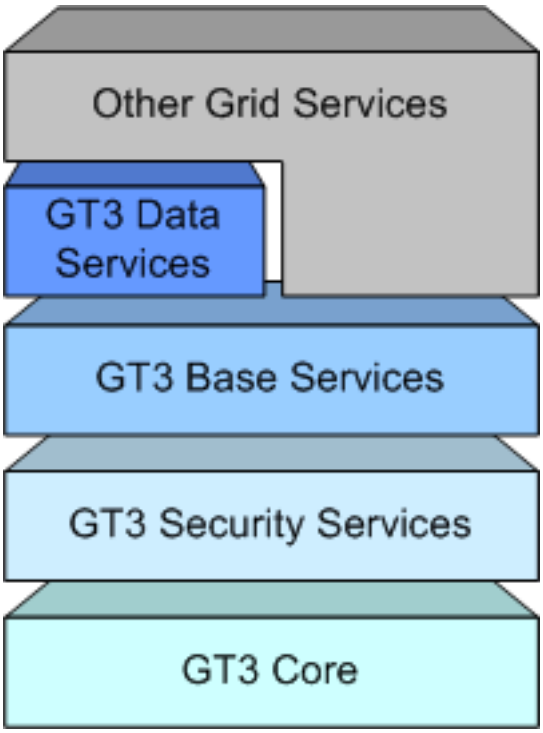
# The Globus Toolkit 3 Programmer's Tutorial

## Key Concepts

### The GT3 Architecture

**<-- Previous ^Up^ Next -->**

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.




Grid Services, which we have already seen, are 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 and, of course, we want all the data exchanged between MathService and clients to be encrypted so we can keep malicious users from intercepting our data. 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'), The Managed Job Service allows us to treat our invocations like jobs, and manage them accordingly.
    - **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 is 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 netwok 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.

**<-- Previous ^Up^ Next -->**

*Borja Sotomayor*

# The Globus Toolkit 3 Programmer's Tutorial

## Key Concepts

### Where to learn Java & XML

**<-- Previous ^Up^ Next -->**

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: The official tutorial from Sun, the makers of Java. Very good if you know absolutely nothing about Java.
- The Coffee Break: 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: Tutorial that covers both the basics and the more advanced aspects of XML.
- ZVON.org: Tons of XML resources. Includes some very good reference guides.

**<-- Previous ^Up^ Next -->**

*Borja Sotomayor*

# The Globus Toolkit 3 Programmer's Tutorial

## Installation

**<-- Previous ^Up^ Next -->**

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 Alpha Installation (for Redhat 7.x): Complete installation guide. Includes information on configuring certificates, databases, and MMJFS. Written by Xin Ling.
- Step-By-Step Example for the Globus Toolkit 3.0: Includes a pretty thorough section on installation. Written by Javier Cano.
- Grid Install for Windows 2000 Platform: Excellent starting point for Windows users. Includes plenty of screenshots. Written by Michael Schneider.
- Installing and deploying GT3 on WebSphere Application Server V5.0 (requires registration)

**<-- Previous ^Up^ Next -->**

*Borja Sotomayor*

# The Globus Toolkit 3 Programmer's Tutorial

## Writing Your First Grid Service

**<-- Previous ^Up^ Next -->**

In this first section we are going to write and deploy a simple Grid Service. Well, that's not totally true: we're actually going to write a basic Web Service, without using any of the OGSA extensions mentioned earlier. However, we'll deploy the service using the GT3 tools, so you can get to know them a little better. Even if you have experience with Web Services (and Apache Axis) you should take a look at this section to get more aquainted with GT3.

Our first Grid Service, which we will use for the rest of the tutorial, is actually the *Math Grid Service* mentioned in the What is a Grid Service? and The GT3 Architecture sections. Of course, our Math Grid Service (from now on, we'll refer to it as *MathService*) won't need a cluster, and will only provide the following simple operations:

- Addition
- Subtraction
- Multiplication
- Division

High-tech stuff, huh? Don't worry, in the following sections we'll gradually improve MathService to make it look more like a real Grid Service, by adding things like notifications, persistency, job management, security, etc.

Since this is probably your very first Grid Service, we're going to do everything step-by-step, so you can get a clear idea of what's involved in writing and deploying a Grid Service. However, don't be discouraged by all the different steps involved in the process. You won't have to repeat them *every time* you want to write a Grid Service. At the end of this section we'll see there's a wonderful tool called `Ant` that allows you to automatically deploy a Grid Service with just one command. Also, remember that the example we are about to see is actually a *plain Web Service*. It is intended for you to get acquainted with GT3 tools and the whole process involved in creating a Grid Service, but it *is not* representative of Grid Service programming with the GT3. We will begin seeing *real* Grid Services in the (appropriately titled) Writing a real Grid Service :-)

Ready to start? Ok! First of all, create an empty directory to put all the files mentioned in this example. All the paths in the example will be relative to that empty directory. We will also refer to it as `$TUTORIAL_DIR`.

**<-- Previous ^Up^ Next -->**

*Borja Sotomayor*

# The Globus Toolkit 3 Programmer's Tutorial

## Writing Your First Grid Service

### Defining the Service Interface

**<-- Previous ^Up^ Next -->**

The first step in writing a Web Service (or a Grid 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. The service interface is also called the *port type* in Web Services jargon (usually written *PortType*).

Remember from A short introduction to Web Services that 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, we need to write a description of our MathService using WSDL. Actually 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 an interface language**. We can generate WSDL automatically from a Java interface or an IDL interface. This is the easiest option, but not the most versatile (since very complicated interfaces are not always converted correctly to WSDL).

We will soon find out (in the GWSDL interface description section) that writing WSDL directly has many advantages, and is generally the preferred way of describing a Grid Service. However, for now we'll go with the easy option: describing our Grid Service using a Java interface and using that to generate the WSDL code.

```
package gt3tutorial.core.first.impl;

public interface Math
{
        public int add(int a, int b);

        public int subtract(int a, int b);

        public int multiply(int a, int b);

        public float divide(int a, int b);
}
```

Save this file as `Math.java` in a new directory named `$TUTORIAL_DIR/gt3tutorial/core/first/impl/` (notice that we're including this interface in the `gt3tutorial.core.first.impl` package). Now that we have the Java interface, we're ready to generate the MathService WSDL.

**<-- Previous ^Up^ Next -->**

*Borja Sotomayor*

# The Globus Toolkit 3 Programmer's Tutorial

## Writing Your First Grid Service

### Our service interface in WSDL

**<-- Previous ^Up^ Next -->**

Generating the WSDL code from the Java interface is a pretty straightfoward process. We'll simply use an Apache Axis tool called `Java2WSDL`. First of all, we have to compile the Java interface:

```
javac Math.java
```

Now we can generate the WSDL. Be sure to run this command from `$TUTORIAL_DIR/`, *not* from `$TUTORIAL_DIR/gt3tutorial/core/first/impl/`. Otherwise, Java2WSDL will fail.

```
java org.apache.axis.wsdl.Java2WSDL
        -P MathPortType
        -S MathService
        -l http://localhost/ogsa/services/core/first/
MathService
        -n http://first.core.gt3tutorial/Math
        gt3tutorial.core.first.impl.Math
```

(Read this if you get a NoClassDefFoundError)

After running this command, you should have a file called `MathService.wsdl` in `$TUTORIAL_DIR/`. Take a peek at it; you'll see some familiar names (Add, Subtract, etc.) For the moment it is safe to ignore the parameters we're sending Java2WSDL. Remember you won't have to use this command directly. A tool called `Ant` will make all this process much simpler.

Hold on! We're not finished with WSDL yet. Now we have to *decorate* the WSDL file. 'Decorating the WSDL' basically means we have to add some extra stuff to the WSDL file for it to work with GT3. We won't use Apache Axis to do this. We'll use a GT3 tool called `DecorateWSDL`.

```
java org.globus.ogsa.tools.wsdl.DecorateWSDL
        $GLOBUS_DIRECTORY/schema/ogsi/ogsi_bindings.
wsdl
        MathService.wsdl
```

($GLOBUS_DIRECTORY should point to the root of your GT3 installation)

Now we finally have a WSDL file that accurately describes our Grid Service. So far, we've only said that our Grid Service can add, subtract, multiply, and divide. We've said *what* it can do, but we haven't specified *how* it will do it. The logical next step is to actually *implement* our Grid Service. However, there's still one more thing we have to do first: generate the stubs.

**<-- Previous ^Up^ Next -->**

*Borja Sotomayor*

# The Globus Toolkit 3 Programmer's Tutorial

## Writing Your First Grid Service

### Generating the Stubs

**<-- Previous ^Up^ Next -->**

If you've worked with RMI, CORBA, EJBs, or any other distributed computing technology, you should have a clear idea of what stubs are. If you don't, now's a good time to reread A short introduction to Web Services. Stubs basically do all the dirty work. They're in charge of all the SOAP and network communication, so us programmers can concentrate on the important stuff.

We can generate all necessary stubs (both the client-side and the server-side one) by running the following command.

```
java org.globus.ogsa.tools.wsdl.GSDL2Java
MathService.wsdl
```

If you take a peek at the `$TUTORIAL_DIR/gt3tutorial/core/first/impl/` directory, you'll notice there's a new directory called `Math`. This directory contains the stubs. They're Java classes which we will be using in the following sections. You'll also notice that there's a `$TUTORIAL_DIR/org/` directory. You can ignore it, but don't erase it.

**<-- Previous ^Up^ Next -->**

*Borja Sotomayor*

# The Globus Toolkit 3 Programmer's Tutorial

## Writing Your First Grid Service

### Implementing your service

**<-- Previous ^Up^ Next -->**

Now we're finally ready to implement our service. The implementation of a Grid Service is simply a Java class which has to meet certain requirements. In this class we'll provide an implementation for the methods we specified in the service interface (`Math.java`). We can also provide additional *private* methods which won't be available through the Grid Service interface, but which our service can use internally.

We'll put the implementation of our Grid Service in the same directory as the service interface: `$TUTORIAL_DIR/gt3tutorial/first/impl/`. Let's call the new file `MathImpl.java`. We'll start with the usual `package` declaration and `import`s (we have to import some OGSA classes which are necessary to implement a Grid Service).

```
package gt3tutorial.core.first.impl;

import org.globus.ogsa.impl.ogsi.GridServiceImpl;
import gt3tutorial.core.first.Math.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).
- 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 provides a particular PortType to the outer world: the Math PortType. But, where did `MathPortType` come from? This is one of the files generated in the previous step ("Generating the stubs"). If you take a look at the file `MathPortType.java`, you'll see that it is very similar to the service interface we wrote in the first step (`Math.java`). Like the `Math` interface, the `MathPortType` interface also specifies the methods which our service supplies, but includes

additional code to make it a *remote interface* (an interface which can be accessed remotely, as opposed to locally). In fact, `MathPortType.java` is generated from the WSDL file which, in turn, was generated from `Math.java`.

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

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

Finally, we implement the methods specified in the service interface: add, subtract, multiply, and divide. 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 int add(int a, int b) throws RemoteException
{
        return a + b;
}

public int subtract(int a, int b) throws RemoteException
{
        return a - b;
}

public int multiply(int a, int b) throws RemoteException
{
        return a * b;
}

public float divide(int a, int b) throws RemoteException
{
        return a / b;
}
```

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 gt3tutorial.core.first.impl;

import org.globus.ogsa.impl.ogsi.GridServiceImpl;
import gt3tutorial.core.first.Math.MathPortType;
import java.rmi.RemoteException;
```

```
public class MathImpl extends GridServiceImpl implements
MathPortType
{
        public MathImpl()
        {
        super("Simple Math Service");
        }

        public int add(int a, int b) throws RemoteException
        {
        return a + b;
        }

        public int subtract(int a, int b) throws
RemoteException
        {
        return a - b;
        }

        public int multiply(int a, int b) throws
RemoteException
        {
        return a * b;
        }

        public float divide(int a, int b) throws
RemoteException
        {
        return a / b;
        }
}
```

**<-- Previous ^Up^ Next -->**

*Borja Sotomayor*

# The Globus Toolkit 3 Programmer's Tutorial

## Writing Your First Grid Service

### The deployment descriptor

**<-- Previous ^Up^ Next -->**

Up to this point, we have written all the Java code our Grid Service needs. We have (1) a service interface, (2) a WSDL file, (3) a bunch of stub files, and (4) an implementation of the service. 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 Grid Service's URL 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="tutorial/core/first/MathService"
provider="java:RPC">
                <parameter name="allowedMethods" value="*"/
>
                <parameter name="className"
value="gt3tutorial.core.first.impl.MathImpl"/>
        </service>

</deployment>
```

Save this as `$TUTORIAL_DIR/gt3tutorial/core/first/Math.wsdd`

In the following sections we will write deployment descriptors which are much more complete, and which use GT3-specific functionality. As mentioned before, we are actually writing a plain Web Service, so this deployment descriptor would be equally valid in an Apache Axis installation without GT3.

However, pay special attention at the following parts of the descriptor:

- `name="tutorial/core/first/MathService"`: This specifies the name of our Grid

Service. 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/tutorial/core/first/MathService`**.

- `<parameter name="allowedMethods" value="*"/>`: This tells the web server that all the methods in the service interface should be available publicly.
- `<parameter name="className" value="gt3tutorial.core.first.impl.MathImpl"/>`. This tells the web server what Java class provides an implementation for this particular Grid Service. That way, when the web server recieves a call for the 'add' method on the 'tutorial/core/first/MathService' Grid Service, it knows that the class responsible for dealing with that call is `gt3tutorial.core.first.impl.MathImpl`.

**<-- Previous ^Up^ Next -->**

*Borja Sotomayor*

# The Globus Toolkit 3 Programmer's Tutorial

## Writing Your First Grid Service

### Compiling and deploying step-by-step

**<-- Previous ^Up^ Next -->**

We've finally reached the point when we are ready to compile everything and deploy it into a web server (in our case, the GT3 standalone container). Right now we're going to do it step-by-step. In the following section we will see that this whole process can be done automatically using the `Ant` tool. Remember, you won't have to do all these steps every time you want to write a Grid Service.

First of all, we'll compile the stubs:

```
javac -sourcepath ./ gt3tutorial/core/first/Math/*.
java
```

Next, we compile the implementation:

```
javac -sourcepath ./ gt3tutorial/core/first/impl/*.
java
```

Now, we need to JAR all the stub classes:

```
jar cvf Math-stub.jar gt3tutorial/core/first/Math/*.
class
```

We also need to JAR the implementation class:

```
jar cvf Math.jar gt3tutorial/core/first/impl/*.class
```

Now we have to create a special kind of JAR called a GAR, or Grid ARchive. This is the actual file that we will deploy into the web server. The fact that it's all neatly packaged into one GAR package makes it very easy to program a Grid Service on one computer, and then deploy it in a different computer. Create a directory called `gar/` and copy the following files into it:

```
./gar/Math.jar
./gar/Math-stub.jar
./gar/server-deploy.wsdd
./gar/schema/tutorial/MathService.wsdl
```

Notice how the WSDD file *must* be named `server-deploy.wsdd`. Also, be sure to recreate that same directory structure (the WSDL file must be inside the `schema/tutorial/` directory)

Now, we can create the GAR:

```
jar cvf Math.gar -C gar/ ./
```

This GAR, as mention above, contains all the files and information the web server needs to deploy the Grid Service. We're going to do the actual deployment with the `Ant` tool, because this is one particular step which would be too long if done by hand. However, if you are anxious to know, this command copies parts of the GAR file (such as the WSDL file and the stubs) 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=GAR_PATH
```

`GAR_PATH` is the path where the GAR file is. Supposing you're user `grid`, and that the GAR file is in your home directory, the command would be:

```
ant deploy -Dgar.name=/home/grid/Math.gar
```

If everything went smoothly: Congratulations! You just deployed your first Grid Service! Now comes the fun part, when we finally put it to the test.

**<-- Previous ^Up^ Next -->**

*Borja Sotomayor*

# The Globus Toolkit 3 Programmer's Tutorial

## Writing Your First Grid Service

### A simple client

**<-- Previous ^Up^ Next -->**

We're going to test our Grid Service with a very simple command-line client which will call the 'add' method. This client will receive three arguments from the command line:

1. The Grid Service Handle (GSH)
2. First number
3. Second number

Create a new directory called `client/` inside the `gt3tutorial/core/first/` directory. In the new directory, create a file called `MathClient.java`

```
package gt3tutorial.core.first.client;

import gt3tutorial.core.first.Math.MathServiceLocator;
import gt3tutorial.core.first.Math.MathPortType;
import java.net.URL;

public class MathClient
{
        public static void main(String[] args)
        {
                try
                {
                        // Get command-line arguments
                        int a = Integer.parseInt(args[1]);
                        int b = Integer.parseInt(args[2]);
                        URL GSH = new java.net.URL(args
[0]);

                        // Get a reference to the remote
web service
                        MathServiceLocator mathService =
new MathServiceLocator();
                        MathPortType math = mathService.
getMathService(GSH);

                        // Call remote method 'add'
                        int sum = math.add(a,b);

                        // Print result
```

```
                        System.out.println(a + " + " + b +
" = " + sum);
                }catch(Exception e)
                {
                        System.out.println("ERROR!");
                        e.printStackTrace();
                }
        }
}
```

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 factories and notifications, 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 `RemoteException`s.

Let's compile the client:

```
javac -sourcepath ./ gt3tutorial/core/first/client/*.
java
```

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
```

> 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 this: How to setup the GT3 scripts (globus-start-container, ogsi-create-service, ...)

After the container starts up (you should see a couple of debug messages on your terminal), we can run the client:

```
java gt3tutorial.core.first.client.MathClient http://
localhost:8080/ogsa/services/tutorial/core/first/
MathService 3 2
```

> I'm supposing a default GT3 installation, with the standalone container located in `http://localhost:8080/ogsa/services`. You may have to modify the URL if you've changed the location of the container.

If all goes well, you should see the following:

```
3 + 2 = 5
```

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

**<-- Previous ^Up^ Next -->**

*Borja Sotomayor*

# The Globus Toolkit 3 Programmer's Tutorial

## Compiling and deploying in one step: Ant

**<-- Previous ^Up^ Next -->**

After programming and deploying your first Grid Service, you're probably thinking you'll have to return constantly to this tutorial to remember all the different steps ("Did I have to generate the stubs first and then implement, or the other way around?" "What files did I have to put in the GAR?" etc.) If you're wondering if there's an easier way to do this, don't worry: there is. In this section we are going to learn about a wonderful tool called `Ant` that will make our life much easier.

First of all we'll review the steps we took to create a Grid Service, and then see how `Ant` makes the whole process simpler. Finally, we'll see a very handy `Ant` buildfile which we will use for the remaining examples in the tutorial.

**<-- Previous ^Up^ Next -->**

*Borja Sotomayor*

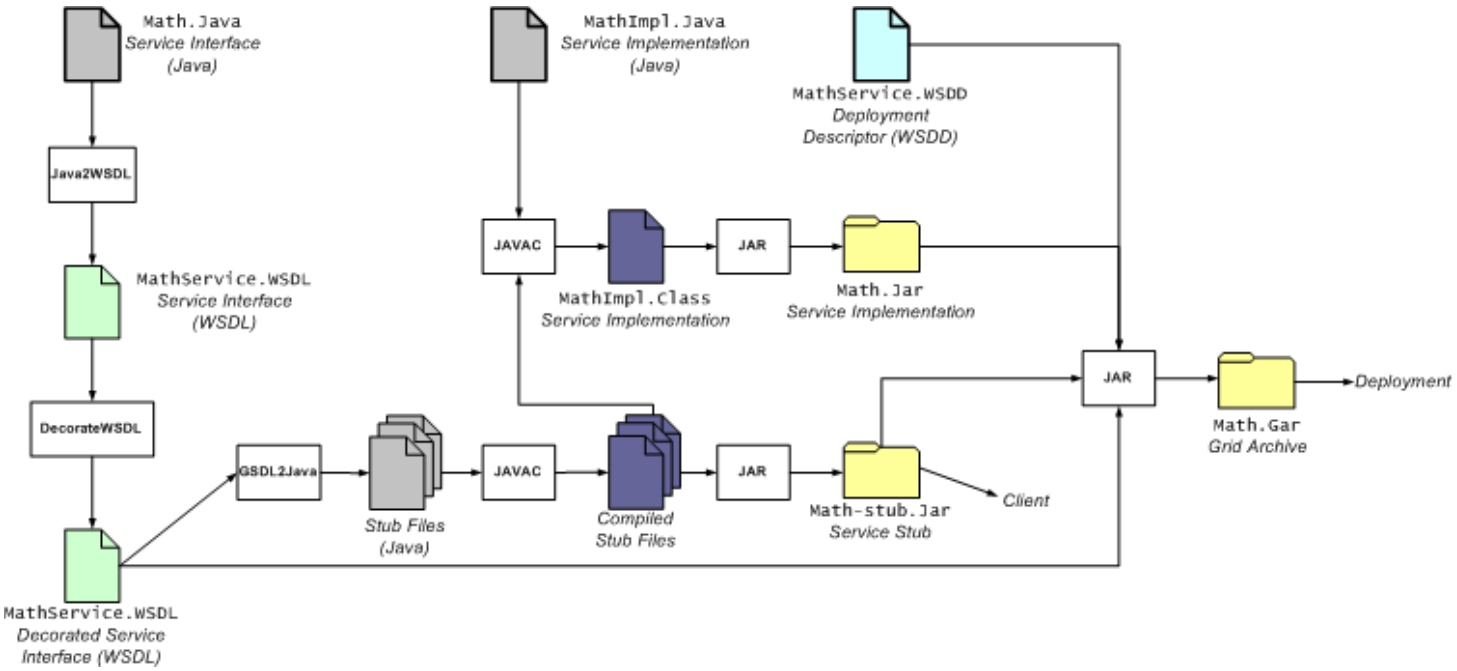
# The Globus Toolkit 3 Programmer's Tutorial

## Compiling and deploying in one step: Ant

### The big picture

**<-- Previous ^Up^ Next -->**

The Grid Service we programmed in the previous section was a fairly simple one. However, we had to take quite a lot of steps to implement and deploy it. All those steps are summarized in the big picture:




Files with a thick line are the ones we had to write on our own. The ones with fine lines are generated automatically by one of the programs we saw in the previous step.

Thanks to `Ant`, arriving at the GAR won't involve so many steps (in fact, it will only involve one step). Go to the next page, and you'll see how the 'big picture' turns into the 'small picture'.

**<-- Previous ^Up^ Next -->**

*Borja Sotomayor*

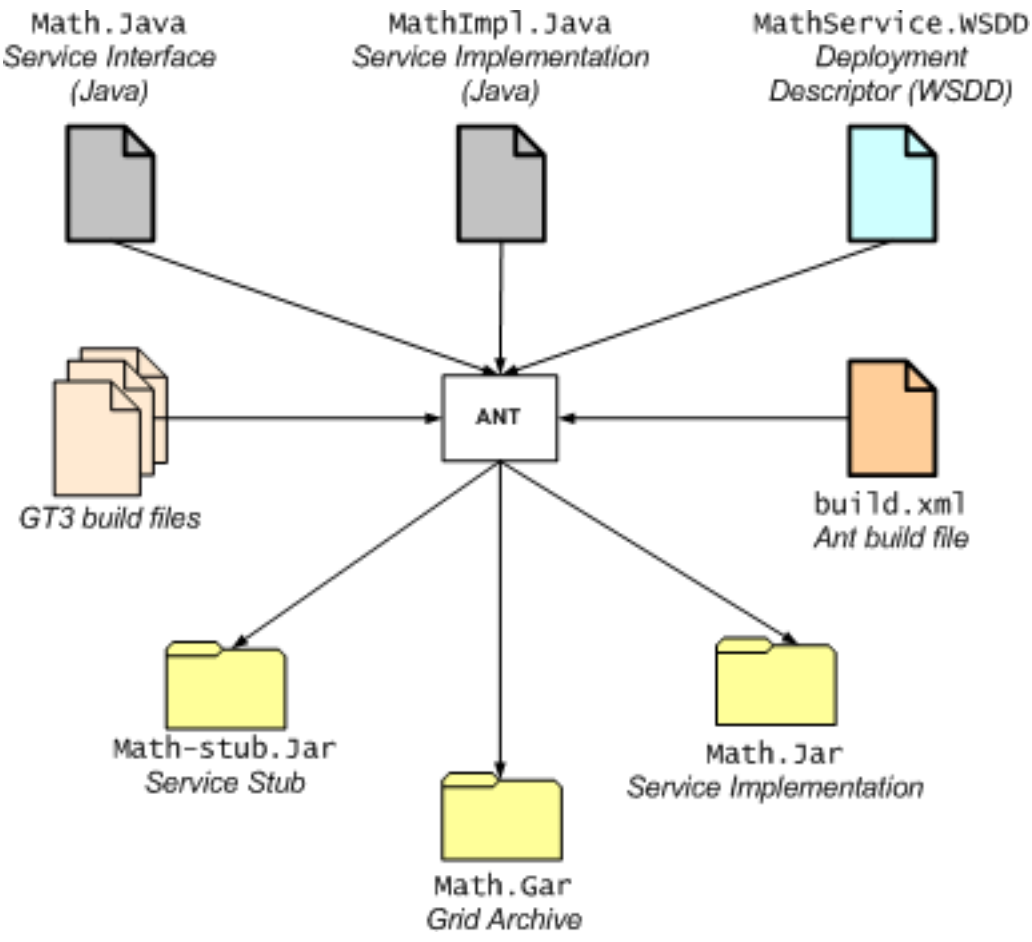
# The Globus Toolkit 3 Programmer's Tutorial

## Compiling and deploying in one step: Ant

### Ant: A Java build tool

**<-- Previous ^Up^ Next -->**

Ant, an Apache Software Foundation project, is a Java *build tool*. In concept, it is very similar to the classic 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 JARs and the GAR directly from the three source files. Internally, it is carrying out all the steps in the big picture, but now we don't have to do 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. It includes plenty of documentation, tutorials, etc.

**<-- Previous ^Up^ Next -->**

*Borja Sotomayor*

# The Globus Toolkit 3 Programmer's Tutorial

## Compiling and deploying in one step: Ant

### Our handy multipurpose buildfile

**<-- Previous ^Up^ Next -->**

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. You can download it here (save it into the directory you're using for the tutorial, $TUTORIAL_DIR). 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.

You also need to 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/`

Now, we're going to try out the build file on our first Grid Service.

```
ant     -Djava.interface=true
        -Dpackage=gt3tutorial.core.first  -
Dinterface.name=Math
        -Dpackage.dir=gt3tutorial/core/first/  -
Dservices.namespace=first.core.gt3tutorial
```

Just by executing this single command, you'll get your JARs and your GAR ready to use in a directory called $TUTORIAL_DIR/build/lib. The only thing left to do is to deploy the GAR.

Hey, wait a second! This might be 'multipurpose', but sure isn't 'handy'! You're right, having to write those four parameter can be a bit of a nuisance. That's why we'll also use a handy shell script which makes things even simpler. You can download it here. With this script, building your Grid Service will be as simple as doing the following:

```
./tutorial_build.sh gt3tutorial/core/first/impl/Math.
java
```

Be sure to run this from $TUTORIAL_DIR.

Of course, you can use this shell script and build file for your own projects. However, bear in mind that they have certain limitations (see this FAQ entry for more details).

P.S.- By the way, I'm not really very good at shell script programming, so if you look at the source code, I'm sure some people will consider it a bit...errr...crude. I'm sure shell gurus could do the same thing with only three lines (and a lot more pipes). Suggestions are more than welcome :-)

**<-- Previous ^Up^ Next -->**

*Borja Sotomayor*

# The Globus Toolkit 3 Programmer's Tutorial

## Writing a Real Grid Service

**<-- Previous ^Up^ Next -->**

Now that we've written a simple Grid Service and know what `Ant` is, we're ready to write a *real* Grid Service. Remember, our first MathService wasn't *really* a Grid Service. It was actually a simple Web Service which we deployed using GT3 tools. What we're going to do now is 100% Grid Service. Really. No strings attached this time.

This might be a good time to reread What is a Grid Service?, where we saw that Grid Services are actually just 'improved Web Services'. In this section we're going to start by seeing one of the main improvements: *factories*.

**<-- Previous ^Up^ Next -->**

*Borja Sotomayor*

# The Globus Toolkit 3 Programmer's Tutorial

## Writing a Real Grid Service

### The Math Factory

**<-- Previous ^Up^ Next -->**

To understand how factories work, we're going to continue with the MathService example. However, this time we're not going to have one single MathService, but a *MathService Factory* that can create multiple *MathService Instances*. These instances are the ones that will peform operations for the clients. In this section we are going to see two ways of using these instances:

- One instance for each client. In this case, each client will work with its own instance, so no one else can access his information. This could be called a stateful transient approach. *Stateful* means that the Grid Service remembers information from one call to the next. *Transient* means that an instance will eventually be destroyed, usually by the client when it has no use for it. In one of the examples we'll see, clients will be able to explicitly tell the Grid Service instances to self-destruct.
- One instance shared by several clients: In this case, all the clients will have access to the information contained in the Grid Service. This is a stateful non-transient approach. *Non-transient* means that the instance is not outlived by its clients (i.e. the instance isn't destroyed when a client finishes with it, because other clients might need to access it). Usually some central authority will be in charge of creating and destroying the instances.

> NOTE: Sometimes the term 'persistent' is used as the opposite of 'transient'. I am using 'non-transient' so you won't get confused when, later on, we see a type of persistency which isn't related to transient/non-transient.

The "1 instance, 1 client" approach is probably the most interesting novelty in Grid Services (with respect to Web Services), since the other approach can be simulated (to some extent) by plain Web Services. However, when we see notifications in the next section, we will see that the "1 instance, N clients" also has its advantages.

Defining a MathService Factory is very similar to the simple MathService we saw earlier. We still need three fundamental files:

- The service interface: Directly in WSDL, or generated from a Java interface.
- The service implementation: Written in Java.
- The deployment descriptor: Written in WSSD.

### The Service Interface

We're going to make a couple changes in the service interface of MathService, but only so we can see

the advantages of Grid Services better. First of all, to keep the example simple, we're getting rid of the `multiply` and `divide` methods (you can add them later on as an exercise). Another big change is that the remaining `add` and `subtract` method will have only one argument, and won't return any value. We're also adding a new method called `int getValue()`. This will be explained shortly.

```
package gt3tutorial.core.factory.impl;

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

        public void subtract(int a);

        public int getValue();
}
```

Save this file as $TUTORIAL_DIR/gt3tutorial/core/factory/impl/Math.java

## The Service Implementation

Take a look at the implementation code, and try to figure out what it does. Then, read the comments below, which explain exactly why the `add` and `subtract` receive only one argument (instead of two, which seems like the more logical choice).

```
package gt3tutorial.core.factory.impl;

import org.globus.ogsa.impl.ogsi.GridServiceImpl;
import gt3tutorial.core.factory.Math.MathPortType;
import java.rmi.RemoteException;

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

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

        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;
        }
}
```

Save this file as $TUTORIAL_DIR/gt3tutorial/core/factory/impl/MathImpl.java

After reading the implementation code, you've probably figured out what has changed in this MathService: it is a stateful service (unlike our first MathService which was stateless). The Grid Service now has a private attribute called `value` which has an initial value of zero. `add` and `subtract` work with that internal value along with the parameter sent in the call. Notice how `add` and `subtract` don't return any value. The only way of checking the current internal value is by calling the `getValue` method. This will help us see the statefulness of Grid Service in action (remember, a normal Web Service would be unable to remember an 'internal value').

## The Deployment Descriptor

The deployment descriptor is the file that has the most 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="tutorial/core/factory/
MathFactoryService" provider="Handler" style="wrapped">
                <parameter name="name"
                        value="MathService Factory"/>
                <parameter name="instance-name"
                        value="MathService Instance"/>
                <parameter name="instance-schemaPath"
                        value="schema/gt3tutorial.core.
factory/Math/MathService.wsdl"/>
                <parameter name="instance-baseClassName"
                        value="gt3tutorial.core.factory.
impl.MathImpl"/>

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

```
RPCURIProvider"/>
                <parameter name="factoryCallback"
                        value="org.globus.ogsa.impl.ogsi.
DynamicFactoryCallbackImpl"/>
                <parameter name="operationProviders"
                        value="org.globus.ogsa.impl.ogsi.
FactoryProvider"/>
        </service>

</deployment>
```

Save this file as $TUTORIAL_DIR/gt3tutorial/core/factory/Math.wsdd

As you can see, this deployment descriptor has much more parameters than our first deployment descriptor. The last 8 parameters are usually the same for all Grid Services, and rarely have to be modified (when we see notifications, we'll see that two of those parameters have to be changed so the Grid Service can support notifications).

The parameters that will usually change from one Grid Service to the other are the first four:

- `name`: Description of the Grid Service
- `instance-name`: Description of a Grid Service instance
- `instance-schemaPath`: The path where the WSDL file can be found. In the tutorial, we'll be using the following general path for the WSDL files: `schema/`*name-of-package*`/`*name-of-PortType*`/`*name-of-PortType*`Service.wsdl` However, don't worry too much about this. Our handy multipurpose buildfile and script will take care of putting every file in the right directory.
- `instance-baseClassName`: Base class of the Grid Service instances. In this case, this is the class `MathImpl` we've just seen.

**<-- Previous ^Up^ Next -->**

*Borja Sotomayor*

# The Globus Toolkit 3 Programmer's Tutorial

## Writing a Real Grid Service

### Deploying the Grid Service

**<-- Previous ^Up^ Next -->**

Ok, now let's deploy our MathService Factory. First of all, let's build the Grid Service using our handy buildfile and script:

```
./tutorial_build.sh gt3tutorial/core/factory/impl/
Math.java
```

Run from $TUTORIAL_DIR/

Now, deploy the GAR file:

```
ant deploy -Dgar.name=$TUTORIAL_DIR/build/lib/
gt3tutorial.core.factory.Math.gar
```

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

Start the services container:

```
globus-start-container
```

Remember you need to setup the GT3 command-line clients for this command to work. If you haven't done this, take a look at the following page: How to setup the GT3 scripts (globus-start-container, ogsi-create-service, ...)

**<-- Previous ^Up^ Next -->**

*Borja Sotomayor*

# The Globus Toolkit 3 Programmer's Tutorial

## Writing a Real Grid Service

### Writing a factory client #1

**<-- Previous ^Up^ Next -->**

Now we're going to test the MathService Factory with our own client. This first client will connect to an existing instance, which we can create from the Service Browser, or using a GT3 command we will see later. This client will receive two arguments from the command line

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

Here is the whole source code for the client. You'll notice that it is very similar to the client we wrote for our first MathService. In fact, the only difference is that we're making two calls to MathService: one call to `add` to add a number, and a call to `getValue` to see what the internal value of MathService is.

```
package gt3tutorial.core.factory.client;

import gt3tutorial.core.factory.Math.
MathServiceGridLocator;
import gt3tutorial.core.factory.Math.MathPortType;
import java.net.URL;

public class MathClient
{
        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.getMathService(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();
                }
        }
}
```

Save this file as $TUTORIAL_DIR/gt3tutorial/core/factory/client/MathClient.java

Let's compile the client:

```
javac -classpath ./build/classes:$CLASSPATH
        gt3tutorial/core/factory/client/MathClient.
java
```

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

```
ogsi-create-service
        http://localhost:8080/ogsa/services/tutorial/
core/factory/MathFactoryService
        math
```

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 page: How to setup the GT3 scripts (globus-start-container, ogsi-create-service, ...)

Notice how ogsi-create-service receives two arguments: the MathService factory GSH and the name of the instance we want to create.

Now you can go ahead and test the client:

```
java gt3tutorial.core.factory.client.MathClient
     http://localhost:8080/ogsa/services/tutorial/
core/factory/MathFactoryService/math
     5
```

If you run the client several times, you'll see how MathService's internal value changes after each call.

**<-- Previous ^Up^ Next -->**

*Borja Sotomayor*

# The Globus Toolkit 3 Programmer's Tutorial

## Writing a Real Grid Service

### Writing a factory client #2

**<-- Previous ^Up^ Next -->**

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 is a simple example of how we can use transient Grid Services with GT3. 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 gt3tutorial.core.factory.client;

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

import gt3tutorial.core.factory.Math.
MathServiceGridLocator;
import gt3tutorial.core.factory.Math.MathPortType;

import java.net.URL;

public class MathClient2
{
        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
                        // to its Math PortType.
                        LocatorType locator = mathFactory.
createService();
                        MathServiceGridLocator mathLocator
=   new MathServiceGridLocator();
                        MathPortType math = mathLocator.
getMathService(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);

                        // Get a reference to the
GridService PortType
                        // and destroy the instance
                        GridService gridService =
gridLocator.getGridServicePort(locator);
                        gridService.destroy();
                }catch(Exception e)
                {
                        System.out.println("ERROR!");
                        e.printStackTrace();
                }
        }
}
```

Save this file as $TUTORIAL_DIR/gt3tutorial/core/factory/client/MathClient2.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. The following lines show how we can create a new MathService instance with the 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 a special PortType that *all* Grid Services have: the GridService PortType. This PortType has a set of useful methods which we will discover throughout the tutorial. The method we're interested in now is `destroy()`, which orders a Grid Service to destroy itself.

```
GridService gridService = gridLocator.getGridServicePort
(locator);
gridService.destroy();
```

Notice how, before making the `destroy()` call, we need to get a reference to the GridService PortType (using the `getGridServicePort` method in the `OGSIServiceGridLocator` object we created earlier).

Let's give this new client a try:

```
javac -classpath ./build/classes:$CLASSPATH
        gt3tutorial/core/factory/client/MathClient2.
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 gt3tutorial.core.factory.client.MathClient2
        http://localhost:8080/ogsa/services/tutorial/
core/factory/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.

**<-- Previous ^Up^ Next -->**

*Borja Sotomayor*

# The Globus Toolkit 3 Programmer's Tutorial

## GWSDL interface description

**<-- Previous ^Up^ Next -->**

In the two examples we've done so far, we've seen that the very first step in writing a Grid Service is *defining the service interface*. In the end, this service interface will have to be WSDL, since that is the language used in the Web Services architecture to describe service interfaces. However, in our very first example we saw that it wasn't strictly necessary to directly write the WSDL code. We could start out with a Java interface, and generate the WSDL code from the Java code. In fact, that what we've done in the previous examples.

However, although writing a Java interface is certainly simple, it does have its shortcomings. Writing WSDL code directly might seem more dificult but, in the long run, is definitely the best choice.

In this section we'll write and deploy a Grid Service with the same functionality seen in the previous section (add, subtract, and getValue), but we'll start with a WSDL description, not with a Java interface. Furthermore, we'll introduce a special kind of WSDL used currently in the Globus Toolkit: the Grid WSDL (or GWSDL).

**<-- Previous ^Up^ Next -->**

*Borja Sotomayor*

# The Globus Toolkit 3 Programmer's Tutorial

## GWSDL interface description

### Java interface versus WSDL description

**<-- Previous ^Up^ Next -->**

Before starting to write code, let's take a look at why WSDL (and GWSDL) are better than starting from an interface language (such as Java or IDL).

- **Semantically richer**: You can be much more 'expressive' with WSDL than with interface languages. In other words, you might find that there are certain operations or requirements that you are unable to express in Java, which you *can* express in WSDL.
- **More control**: Starting out from an interface language means that, at some point, you'll have to run some *translation* program that will turn your Java code to WSDL code. And let's face it: each time you add a new layer of abstraction over anything (be it software or hardware) you lose a bit of control over the final result, despite making your job easier. This means that the final WSDL code might not be exactly what you need it to be.

Of course, interface languages do have an important advantage over WSDL: they're *much* easier to write. A 10 line Java interface can easily turn into a 50-100 line WSDL document (which, being an XML language, includes a wide assortment of tags, attributes, namespaces,... fun!).

Even so, don't let yourself be intimidated by WSDL. When you see your first WSDL document in the next page, you might think of saying *"No way! I'm sticking with Java!"*, but you'll eventually realize that writing WSDL is not so difficult after all. Besides, the rest of the tutorial examples start from GWSDL descriptions, so you don't have much of a choice, do you? :-)

**<-- Previous ^Up^ Next -->**

*Borja Sotomayor*

# The Globus Toolkit 3 Programmer's Tutorial

## GWSDL interface description

### Writing the GWSDL description

**<-- Previous ^Up^ Next -->**

As mentioned before, the Grid Service we are going to write and deploy is practically the same as the one seen in the previous section, except we're starting with a WSDL description, not a Java interface like this:

```
package gt3tutorial.core.gwsdl.impl;

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

        public void subtract(int a);

        public int getValue();
}
```

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). In a moment we'll see the 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 page of the **How to...** section: How to write a GWSDL description of your Grid Service.

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

```
<?xml version="1.0" encoding="UTF-8"?>
<definitions name="MathService"
        targetNamespace="http://www.gt3tutorial.org/
namespaces/0.2/core/gwsdl/Math"
        xmlns:tns="http://www.gt3tutorial.org/
namespaces/0.2/core/gwsdl/Math"
```

```
        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.gt3tutorial.org/
namespaces/0.2/core/gwsdl/Math"
        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>
```

Save this file as $TUTORIAL_DIR/gt3tutorial/core/gwsdl/schema/Math.gwsdl

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.gt3tutorial.org/namespaces/0.2/core/gwsdl/Math
```

Furthermore, we have to include 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. We don't have to add them manually, since they are generated automatically with a GT3 tool. We won't even have to use that tool manually, since it is a part of the build process defined in the GT3 `Ant` build files (so each time we want to generate the stubs for our Grid Service starting from a GWSDL file, the bindings will be generated automatically for us).

## 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. Since WSDL 1.2 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 was unable to use WSDL 1.2 (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 is a W3C Recommendation (a stable standard), it will substitute GWSDL for WSDL 1.2.

So, what exactly are the improvements in GWSDL (and, therefore, in WSDL 1.2)? Well, if you are WSDL-literate, you have probably 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/WSDL 1.2: PortType inheritance. 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).

Of course, you might be wondering why we haven't seen the GridService interface before. We'll, remember how in the previous example our `MathImpl` class extended from a base class called

`GridServiceImpl`? Well, *that* class implements the GridService interface.

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

**<-- Previous ^Up^ Next -->**

*Borja Sotomayor*

# The Globus Toolkit 3 Programmer's Tutorial

## GWSDL interface description

### Mapping the namespaces to packages

**<-- Previous ^Up^ Next -->**

We're done defining the service interface, so the next step is to implement the Grid Service. However, before doing that, we have to do a small (but important) task. Remember that we said that the GWSDL code would be equivalent to the following Java interface?

```
package gt3tutorial.core.gwsdl.impl;

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

        public void subtract(int a);

        public int getValue();
}
```

Sure, our GWSDL specifies those three operations, with all their parameters and return values. But what about the package name? During the build process, a set of stubs will be generated which we will need to use both in the server side and the client side. But there is no mention whatsoever in the GWSDL file of what Java package those stubs will belong to. In fact, the default behaviour is to generate the package name from the target namespace of the GWSDL file. Left to itself, the stub generator would create the following package:

```
package org.gt3tutorial.www.namespaces.0.2.core.gwsdl.Math;
```

Or something equally hideous. Fortunately, we can tell the stub generator to *map* certain namespaces to specific package names by creating a *mapping file*:

```
http\://www.gt3tutorial.org/namespaces/0.2/core/gwsdl/
Math=gt3tutorial.core.gwsdl.wsdl
http\://www.gt3tutorial.org/namespaces/0.2/core/gwsdl/Math/
bindings=gt3tutorial.core.gwsdl.wsdl.bindings
http\://www.gt3tutorial.org/namespaces/0.2/core/gwsdl/Math/
service=gt3tutorial.core.gwsdl.wsdl.service
```

Save this file as $TUTORIAL_DIR/namespace2package.mappings

Yes, the backslash before the colon (`http\://...`) *is* intentional.

The first namespace is the target namespace of the GWSDL file. The other two namespaces (our namespace plus `/bindings` and `/service`) are created by the stub generator, so we have to define a mapping for them too.

**<-- Previous ^Up^ Next -->**

*Borja Sotomayor*

# The Globus Toolkit 3 Programmer's Tutorial

## GWSDL interface description

### Implementing the Grid Service

**<-- Previous ^Up^ Next -->**

We're finally ready to implement the Grid Service. Since its behaviour is exactly the same as the previous example, the implementation is practically the same:

```
package gt3tutorial.core.gwsdl.impl;

import org.globus.ogsa.impl.ogsi.GridServiceImpl;
import gt3tutorial.core.gwsdl.wsdl.MathPortType;
import java.rmi.RemoteException;

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

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

        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;
        }
}
```

Save this file as $TUTORIAL_DIR/gt3tutorial/core/gwsdl/impl/MathImpl.java

There is, however, a small difference. In our previous example, the MathPortType Java interface

belonged to the following package:

```
gt3tutorial.core.factory.Math
```

However, we mapped the WSDL target namespace to the following package:

```
gt3tutorial.core.gwsdl.wsdl
```

So, basically, the stub classes are no longer stored in a subpackage named after the original Java interface (`Math`), but in the package we explicitly specify in the mappings file.

## The Deployment Descriptor

The deployment descriptor is also practically the same as the one in the previous example.

```
<?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/core/gwsdl/
MathFactoryService" provider="Handler" style="wrapped">
                <parameter name="name"
                        value="MathService Factory
(GWSDL)"/>
                <parameter name="instance-name"
                        value="MathService Instance
(GWSDL)"/>
                <parameter name="instance-schemaPath"
                        value="schema/gt3tutorial.core.
gwsdl/Math/Math_service.wsdl"/>
                <parameter name="instance-baseClassName"
                        value="gt3tutorial.core.gwsdl.impl.
MathImpl"/>

                <!-- Start common parameters -->
                <parameter name="allowedMethods" value="*"/
>
                <parameter name="persistent" value="true"/>
                <parameter name="className"
                        value="org.gridforum.ogsi.Factory"/
>
                <parameter name="baseClassName"
                        value="org.globus.ogsa.impl.ogsi.
PersistentGridServiceImpl"/>
                <parameter name="schemaPath"
                        value="schema/ogsi/
ogsi_factory_service.wsdl"/>
```

```
                <parameter name="handlerClass"
                        value="org.globus.ogsa.handlers.
RPCURIProvider"/>
                <parameter name="factoryCallback"
                        value="org.globus.ogsa.impl.ogsi.
DynamicFactoryCallbackImpl"/>
                <parameter name="operationProviders"
                        value="org.globus.ogsa.impl.ogsi.
FactoryProvider"/>
        </service>

</deployment>
```

Save this file as $TUTORIAL_DIR/gt3tutorial/core/gwsdl/Math.wsdd

Aside from minor name changes (the package for this Grid Service is called `gwsdl`, not `factory`), there is one noteworthy change: when we start from a GWSDL description, the name of the generated WSDL file will not be *interface-name*`Service.wsdl` (for example, `MathService.wsdl`), but *interface-name*`_service.wsdl` (in this case: `Math_service.wsdl`).

## Compile and Deploy

Our handy multipurpose buildfile and script work just as well when starting from a GWSDL description. Instead of passing the Java interface to the script, we'll pass the GWSDL file:

```
./tutorial_build.sh gt3tutorial/core/gwsdl/schema/
Math.gwsdl
```

Once you've compiled the Grid Service, deploy it:

```
ant deploy -Dgar.name=$TUTORIAL_DIR/build/lib/
gt3tutorial.core.gwsdl.Math.gar
```

Finally, start the Grid Services container, and create an instance of the Grid Service, which we will access from our client.

```
globus-start-container

ogsi-create-service
        http://localhost:8080/ogsa/services/tutorial/
core/gwsdl/MathFactoryService
        math
```

**<-- Previous  ^Up^  Next -->**

*Borja Sotomayor*

# The Globus Toolkit 3 Programmer's Tutorial

## GWSDL interface description

### A simple client

**<-- Previous ^Up^ Next -->**

Once again, since this examples is nearly identical to the previous one, the client we'll use to test the Grid Service is also the same:

```
package gt3tutorial.core.gwsdl.client;

import gt3tutorial.core.gwsdl.wsdl.service.
MathServiceGridLocator;
import gt3tutorial.core.gwsdl.wsdl.MathPortType;
import java.net.URL;

public class MathClient
{
        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();
                }
            }
}
```

Just notice how the stub generator doesn't place *all* the stub classes in one same package. While the `MathPortType` interface is in the following package:

```
gt3tutorial.core.gwsdl.wsdl
```

The `MathServiceGridLocator` is in the following package:

```
gt3tutorial.core.gwsdl.wsdl.service
```

Let's give the client a try. First of all, compile it:

```
javac -classpath ./build/classes:$CLASSPATH
        gt3tutorial/core/gwsdl/client/MathClient.java
```

And run it:

```
java gt3tutorial.core.gwsdl.client.MathClient
        http://localhost:8080/ogsa/services/tutorial/
core/gwsdl/MathFactoryService/math
        5
```

**<-- Previous  ^Up^  Next -->**

*Borja Sotomayor*

# The Globus Toolkit 3 Programmer's Tutorial

## Operation Providers

**<-- Previous ^Up^ Next -->**

In all the examples we've seen up until now, when implementing our Grid Service we had to create a Java class which extended from `GridServiceImpl`, which all Grid Services must extend from. For example:

```
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.

**<-- Previous ^Up^ Next -->**

*Borja Sotomayor*

# The Globus Toolkit 3 Programmer's Tutorial

## Operation Providers

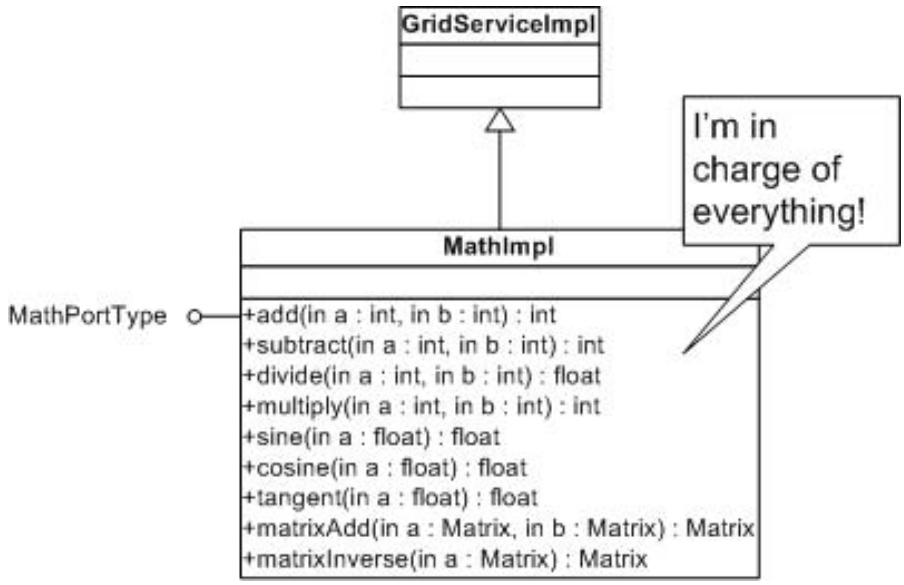
### Inheritance versus Operation Providers

**<-- Previous ^Up^ Next -->**

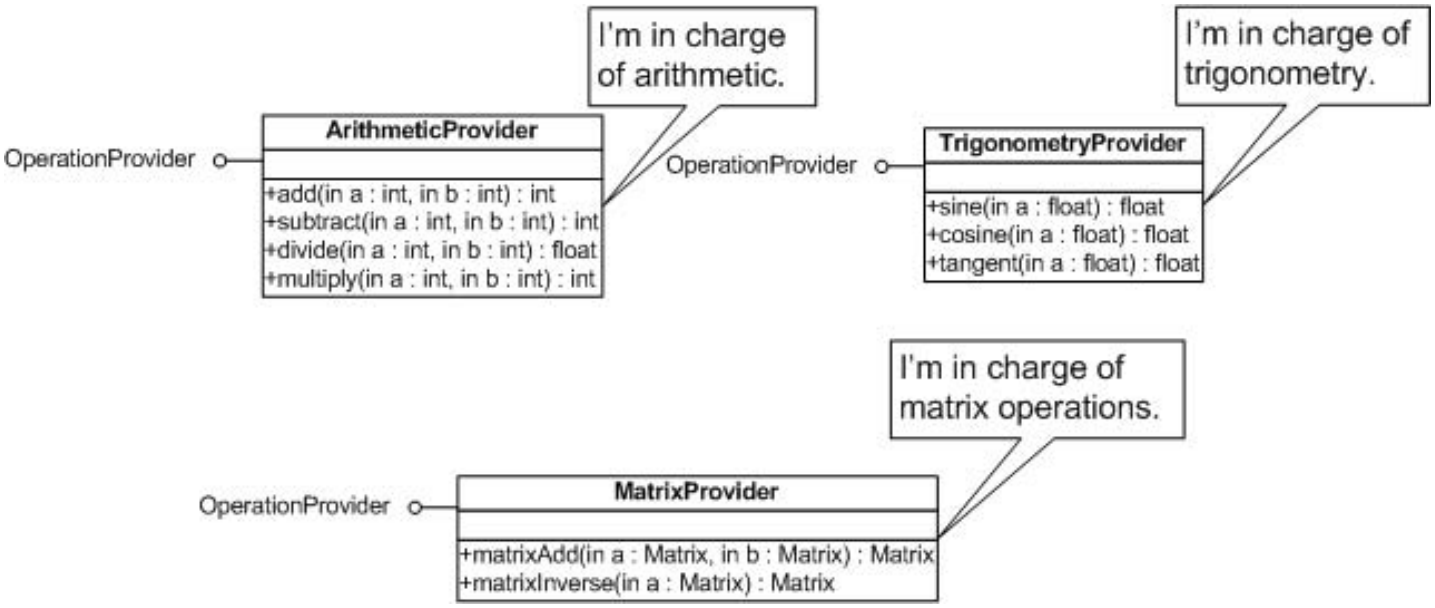
The Globus Toolki3 provides two implementation approaches:

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

The inheritance approach is the one we've seen in all the previous examples. All our Grid Services extend from a class called `GridServiceImpl`, which contains all the basic functionality of a Grid Service. Our class simply has to provide all the operations we want in our Grid Service (all the operations in our MathPortType):

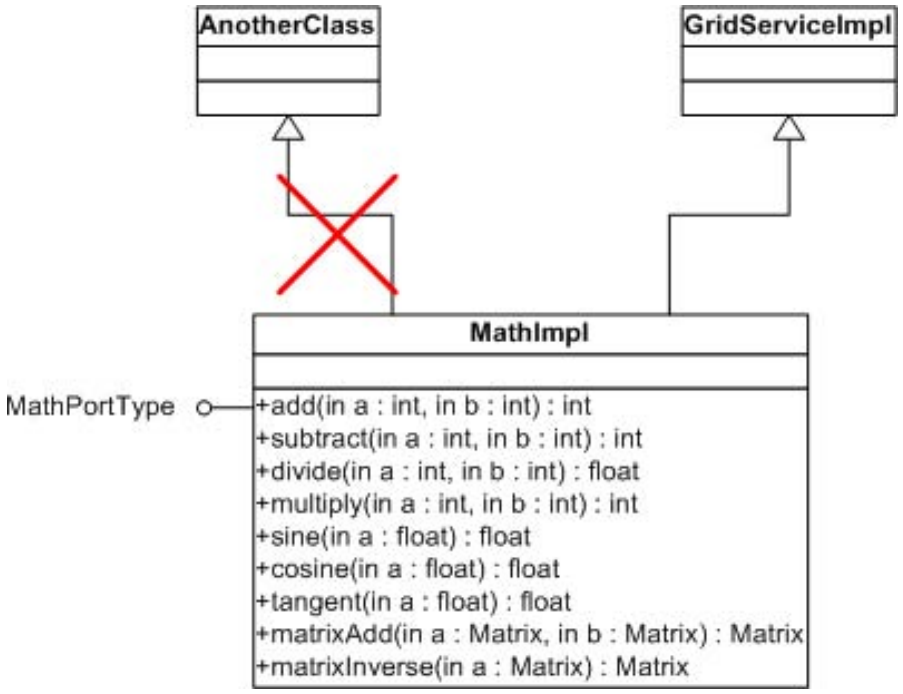



The delegation approach, on the other hand, allows us to distribute the operations of our PortType in several classes. For example, it might make sense to separate the previous `MathImpl` into three classes: one for the arithmetic operations, one for the trigonometry operations, and one for the matrix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

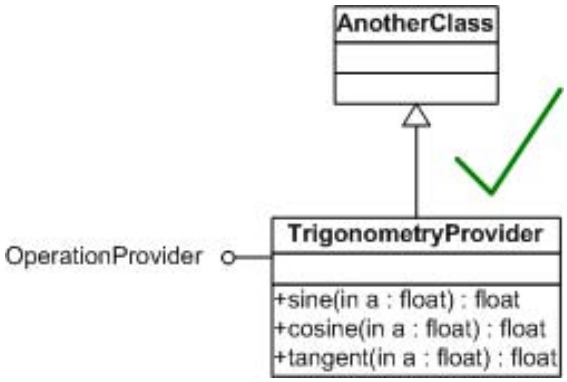



Notice how these classes don't extend from any base class. They only have to implement an interface called `OperationProvider`. Of course, you might be wondering: If there is no base class to extend from, where do these classes get their 'basic Grid Service functionality'? Actually, `GridServiceImpl` 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 page).

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 inheritance 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.

**<-- Previous ^Up^ Next -->**

*Borja Sotomayor*

# The Globus Toolkit 3 Programmer's Tutorial

## Operation Providers

### Writing an operation provider

**<-- Previous ^Up^ Next -->**

The following example, which uses an operation provider instead of extending from `GridServiceImpl`, has the exact same GWSDL file we saw in the previous example (the only difference is the target namespace of the GWSDL file). This means that using the inheritance approach or the delegation approach has no effect whatsoever on the GWDSL description. The GWSDL file for this example is `$TUTORIAL_DIR/gt3tutorial/core/providers/schema/Math.gwsdl`.

The operation provider class (which we will call `MathProvider`) is very similar to all the `MathImpl` classes we saw in the previous examples. 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 gt3tutorial.core.providers.impl;

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

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

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

Save this file as $TUTORIAL_DIR/gt3tutorial/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 two private properties: `base` and `operations`.

```
private static final QName[] operations = new QName[]{new
QName("", "*")};
private GridServiceBase base;
```

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

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

These two methods and properties are basically what make the operation provider fit into the Grid Service. The `operations` property specifies what operations this class *provides*. The Grid Services

container uses the `getOperations()` method to 'ask' our class what operations it provides.

Since our example has only one operation provider, we can use a wildcard in the `operations` property:

```
private static final QName[] operations = new QName[]{new
QName("", "*")};
```

However, if we had more than one operation provider, we would have to list every single method in the `operations` property, like so:

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

Besides those two methods, the rest of the implementation is just like all the previous examples:

```
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;
}
```

**<-- Previous  ^Up^  Next -->**

*Borja Sotomayor*

# The Globus Toolkit 3 Programmer's Tutorial

## Operation Providers

### Deploying the Grid Service

**<-- Previous ^Up^ Next -->**

Let's deploy this Grid Service. First of all, we need to add the following namespace-to-package mappings to our mappings file:

```
http\://www.gt3tutorial.org/namespaces/0.2/core/providers/
Math=gt3tutorial.core.providers.wsdl
http\://www.gt3tutorial.org/namespaces/0.2/core/providers/
Math/bindings=gt3tutorial.core.providers.wsdl.bindings
http\://www.gt3tutorial.org/namespaces/0.2/core/providers/
Math/service=gt3tutorial.core.providers.wsdl.service
```

Add this to file $TUTORIAL_DIR/namespace2package.mappings

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="tutorial/core/providers/
MathFactoryService" provider="Handler" style="wrapped">
                <parameter name="name"
                        value="MathService Factory (with
provider)"/>
                <parameter name="instance-name"
                        value="MathService Instance (with
provider)"/>
                <parameter name="instance-schemaPath"
                        value="schema/gt3tutorial.core.
providers/Math/Math_service.wsdl"/>
                <parameter name="instance-className"
                        value="gt3tutorial.core.providers.
wsdl.MathPortType"/>
                <parameter name="instance-
operationProviders"
                        value="gt3tutorial.core.providers.
```

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

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

</deployment>
```

Save this file as $TUTORIAL_DIR/gt3tutorial/core/providers/Math.wsdd

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

```
<parameter name="instance-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 `instance-operationProviders` parameter:

```
<parameter name="instance-operationProviders"
        value="gt3tutorial.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="instance-className"
        value="gt3tutorial.core.providers.wsdl.
MathPortType"/>
```

Now, let's build the Grid Service:

```
./tutorial_build.sh gt3tutorial/core/providers/
schema/Math.gwsdl
```

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

```
ant deploy -Dgar.name=$TUTORIAL_DIR/build/lib/
gt3tutorial.core.providers.Math.gar

globus-start-container
```

**<-- Previous ^Up^ Next -->**

*Borja Sotomayor*

# The Globus Toolkit 3 Programmer's Tutorial

## Operation Providers

### A simple client

**<-- Previous ^Up^ Next -->**

This client is practically identical to the factory client #2 (the one that created, used, and destroyed an instance). In fact, the only change is the package names. When switching from an inheritance implementation to a delegation implementation, the client is not affected.

```
package gt3tutorial.core.providers.client;

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

import gt3tutorial.core.providers.wsdl.service.
MathServiceGridLocator;
import gt3tutorial.core.providers.wsdl.MathPortType;

import java.net.URL;

public class MathClient
{
        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);

                        // Get a reference to the
GridService PortType
                        // and destroy the instance
                        GridService gridService =
gridLocator.getGridServicePort(locator);
                        gridService.destroy();
                }catch(Exception e)
                {
                        System.out.println("ERROR!");
                        e.printStackTrace();
                }
        }
}
```

Compile the client:

```
javac -classpath ./build/classes:$CLASSPATH
        gt3tutorial/core/providers/client/MathClient.
java
```

And give it a try:

```
java gt3tutorial.core.providers.client.MathClient
        http://localhost:8080/ogsa/services/tutorial/
core/providers/MathFactoryService
        5
```

**<-- Previous  ^Up^  Next -->**

*Borja Sotomayor*

# The Globus Toolkit 3 Programmer's Tutorial

## Logging

**<-- Previous ^Up^ Next -->**

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 component. In this section we'll add logging to the previous example (the Operation Providers example). Even so, the `providers` package (`$TUTORIAL_DIR/core/providers/`) in the examples file doesn't have these modifications. You can find them in a separate package: `$TUTORIAL_DIR/core/logging/`

**<-- Previous ^Up^ Next -->**

*Borja Sotomayor*

# The Globus Toolkit 3 Programmer's Tutorial

## Logging

### The Jakarta Commons Logging architecture

**<-- Previous ^Up^ Next -->**

The goal of the Apache Jakarta Commons Project 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 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 will see in the next page.

**<-- Previous ^Up^ Next -->**

*Borja Sotomayor*

# The Globus Toolkit 3 Programmer's Tutorial

## Logging

### Adding logging to MathService

**<-- Previous ^Up^ Next -->**

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 MathProvider implements OperationProvider
{
        // Create this class's logger
        static Log logger = LogFactory.getLog(MathProvider.
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("Addition 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;
        }
}
```

Add these modifications to $TUTORIAL_DIR/core/providers/impl/MathProvider.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(MathProvider.class.
getName());
```

After these two modifications, our `MathProvider` 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")`

After you've added these modifications, compile and deploy as described in the previous section.

**<-- Previous ^Up^ Next -->**

*Borja Sotomayor*

# The Globus Toolkit 3 Programmer's Tutorial

## Logging

### Viewing log output

**<-- Previous ^Up^ Next -->**

Our `MathProvider` 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_DIRECTORY/ogsilogging.properties`. Just add the following line at the end of that file:

```
gt3tutorial.core.providers.impl.MathProvider=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 `gt3tutorial.core.providers.impl.MathProvider`. 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_DIRECTORY/ogsilogging_parm.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. We'll use the operation providers client from the previous section. Since that client creates and destroys its own instance, we won't have to create it first. Just make sure you've started the Grid Services container before running the client.

```
java gt3tutorial.core.providers.client.MathClient
        http://localhost:8080/ogsa/services/tutorial/
core/providers/MathFactoryService
        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 `gt3tutorial.core.providers.impl.MathProvider`)

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 gt3tutorial.core.providers.client.MathClient
        http://localhost:8080/ogsa/services/tutorial/
core/providers/MathFactoryService
        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 value!
[DATE TIME ] CLASS_NAME [getValue:55] INFO: getValue
() invoked
```

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

```
gt3tutorial.core.providers.impl.MathProvider=console,warn
```

You will need to restart your Grid Services 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 value!
```

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

**<-- Previous ^Up^ Next -->**

*Borja Sotomayor*

# The Globus Toolkit 3 Programmer's Tutorial

## Lifecycle Management

**<-- Previous ^Up^ Next -->**

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.

Most distributed object technologies, including GT3, provide 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.

Once again, we'll work with the Operation Providers example. Remember that the `providers` package (`$TUTORIAL_DIR/core/providers/`) in the examples file doesn't have the modifications we did in the Logging section, and the same applies to the modifications we'll do in this section. You can find them in a separate package: `$TUTORIAL_DIR/core/lifecycle/`

**<-- Previous ^Up^ Next -->**

*Borja Sotomayor*

# The Globus Toolkit 3 Programmer's Tutorial

## Lifecycle Management

### The callback methods

**<-- Previous ^Up^ Next -->**

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 MathProvider implements OperationProvider,
GridServiceCallback
{

        // ...

        // Callback methods
        public void preCreate(GridServiceBase base) throws
GridServiceException
        {
                logger.info("Instance is going to be
created (preCreate)");
        }

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

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

```
        }

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

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

As you can see, the only thing we're doing in the callback methods is writing log messages. If you compile and deploy this modified Grid Service (just as we did in the Operation Providers section), and run the client (which creates and destroys an instance) you'll see some of those messages in the console output of the Grid Services container, informing you that the instance has been created and then destroyed.

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.

**<-- Previous ^Up^ Next -->**

*Borja Sotomayor*

# The Globus Toolkit 3 Programmer's Tutorial

## Lifecycle Management

### The lifecycle monitor

**<-- Previous ^Up^ Next -->**

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 gt3tutorial.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 (preSerializationCall)");
        }

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

This file is $TUTORIAL_DIR/gt3tutorial/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:

```
gt3tutorial.core.lifecycle.impl.
MathLifecycleMonitor=console,info
```

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="gt3tutorial.core.lifecycle.impl.
MathLifecycleMonitor"/>
```

To try this out, just recompile and redeploy the Grid Service (just as we did in the Operation Providers section).

**<-- Previous ^Up^ Next -->**

*Borja Sotomayor*

# The Globus Toolkit 3 Programmer's Tutorial

## Lifecycle Management

### Lifecycle parameters in the deployment descriptor

**<-- Previous ^Up^ Next -->**

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.

**<-- Previous ^Up^ Next -->**

*Borja Sotomayor*

# The Globus Toolkit 3 Programmer's Tutorial

## Service Data

**<-- Previous ^Up^ Next -->**

This section introduces the concept of *Service Data*. After introducing what Service Data is, and what its main uses are in OGSA, we'll see a simple example.

**<-- Previous ^Up^ Next -->**

*Borja Sotomayor*

# The Globus Toolkit 3 Programmer's Tutorial

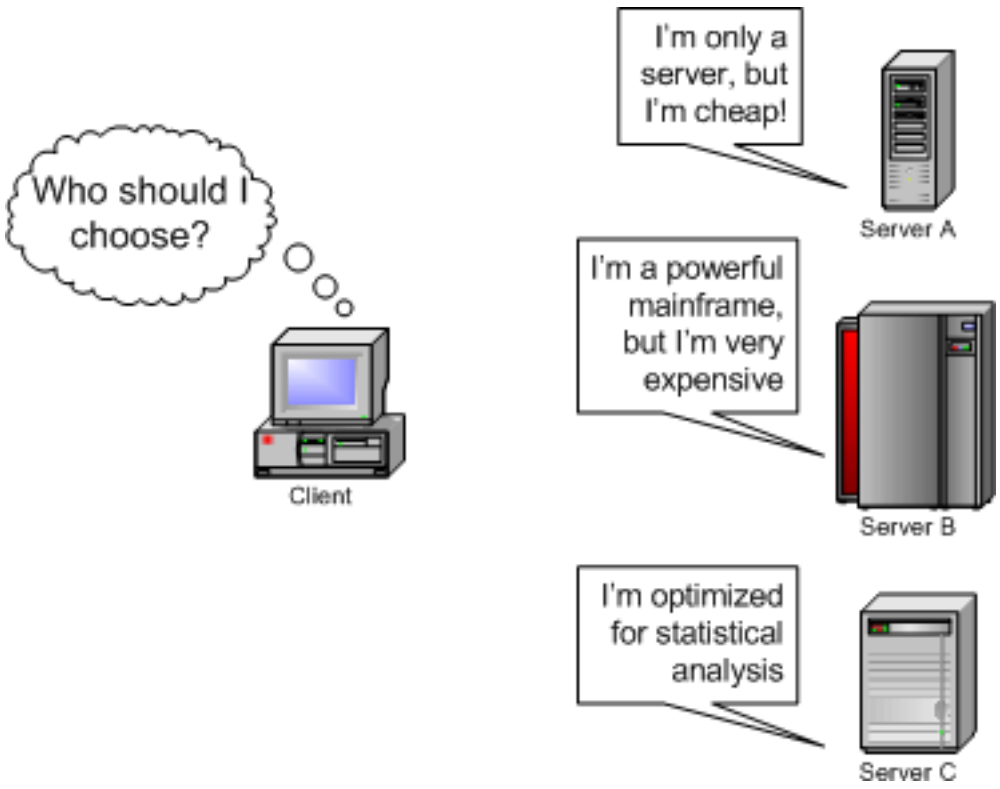
## Service Data

### The logic behind Service Data

**<-- Previous ^Up^ Next -->**

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 by OGSA.

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.

**<-- Previous ^Up^ Next -->**

*Borja Sotomayor*

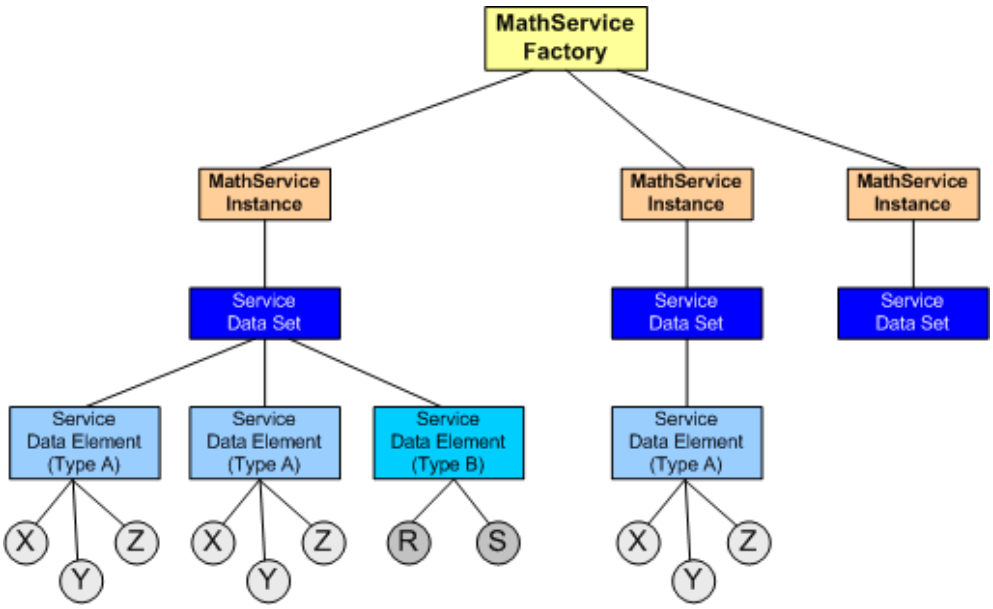
# The Globus Toolkit 3 Programmer's Tutorial

## Service Data

### Service Data in OGSA

**<-- Previous ^Up^ Next -->**

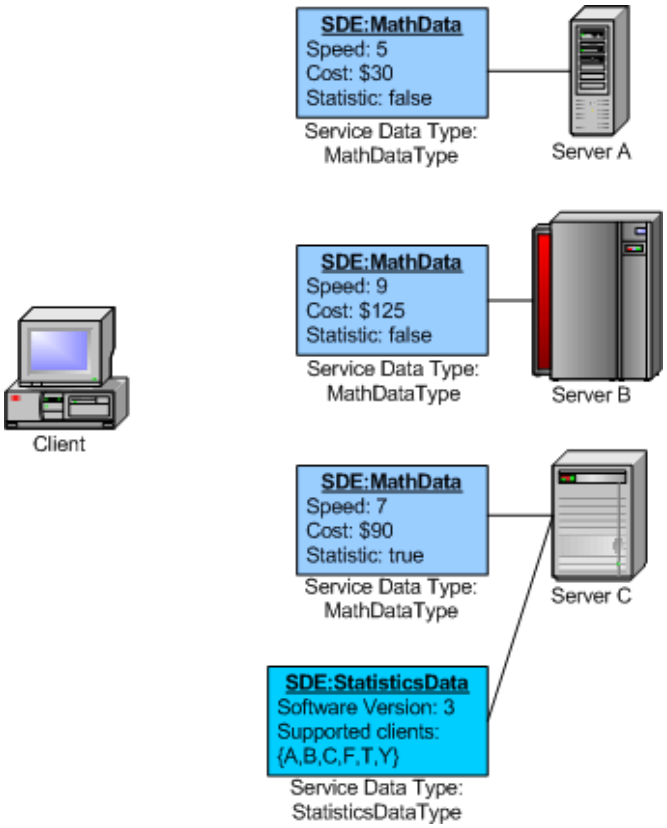
Any Grid Service instance can have Service Data associated to it. The following diagram shows how Service Data is structured in OGSA.




First of all, notice how every instance has a *Service Data Set*. A Service Data Set can contain zero or more Service Data Elements (or SDEs). Each SDE can be of a different type. In the diagram, the first instance has two 'type A' SDEs and one 'type B' SDE. The second instance has only one 'type A' SDE. The third instance has no SDEs at all (even so, it does have an *empty* Service Data Set). Notice how SDEs of the same type always contain the same information ('type A' has X, Y, Z; 'type B' has R and S). The SDEs are the ones that actually contain the data (X, Y, Z, R, S).

If this diagram is too abstract for you, let's take a look at something a little bit more 'real'. Suppose we want to include Service Data in our MathService instances from the previous page. We want all the instances to include data about the 'mathematical characteristics' and, in case they're capable of performing statistical analysis, data about the 'statistical characteristics'. We could define two *types of SDEs*: a `MathDataType` for the mathematical data, and a `StatisticsDataType` for those services with statistics capabilities. `MathDataType` SDEs would include information like the speed and cost of the service. `StatisticsDataType` SDEs would include information like the types of analysis the service is capable of performing.

Our MathService instances could look like this:




In this diagram, we have three MathService instances. Each has a Service Data Set (not shown on the diagram). The first two instances each have one `MathDataType` SDE. Notice how both SDEs contain the same *type* of information (speed, cost, and statistics). The difference between those two SDEs is the values. The third instance has two SDEs: a `MathDataType` and a `StatisticsDataType` one. Notice how the `StatisticsDataType` has a different type of information (software version, 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.

Finally, take a look at the following:

- Each SDE must have a name, which must be *locally unique* (unique inside the instance). All the `MathDataType` SDEs have the same name ("MathData"), but since they're in different instances, that's not a problem. The third instance has two SDEs with different names ("MathData" and "StatisticsData"). The name, of course, allows us to *address* one particular SDE inside the instance.
- An SDE isn't just a list of 'name-value' pairs. It can have complex information structures. For example, the `StatisticsDataType` SDE has an array ("Supported clients", a list of the clients this service supports).

## Service Data & Java Beans

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' properties to the MathService interface? 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 to have a separate Java class for each SDE type. In our example, we would have a class for the `MathDataType` and a class for the `ServiceDataType`. An individual SDE would be an instance of one of those classes.

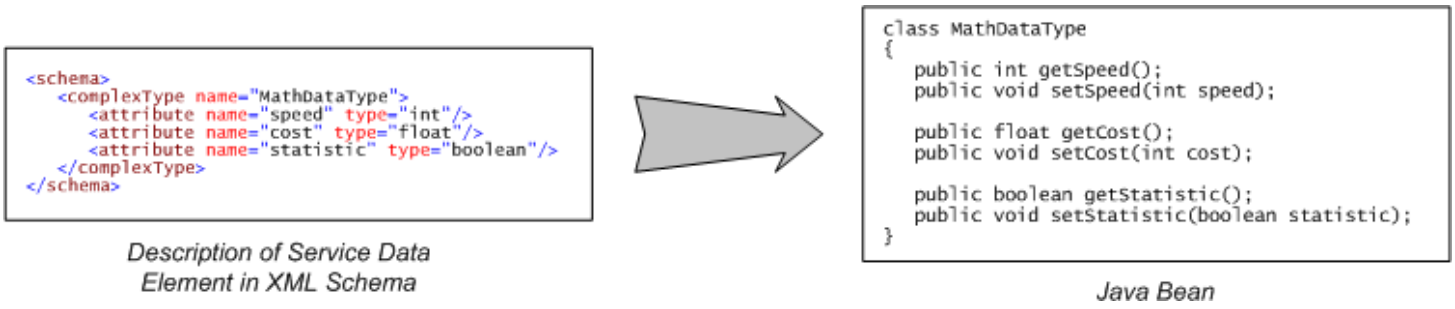
However, these 'SDE classes' are no ordinary Java classes. They have to be Java Beans. If you've programmed a lot with Java, you'll know what I'm talking about. If you're not sure what a Java Bean is, don't worry: just think of them as 'special Java classes' which have to meet a set of requirements. For example, in a Java Bean every attribute must have corresponding get/set methods. Just one thing: don't mistake them with Enterprise Java Beans. That's a completely different kind of bean.

Actually, it doesn't really matter if you know or don't know about Java Beans, because we don't have to write them directly. We'll generate them automatically from a Service Data Description.

## Service Data Description

The Service Data Description (or SDD) describes a 'type of SDE'. For example, the `MathDataType` SDD would specify how this type of SDD must include information about the speed, cost, and 'statistic capability' of the service. This SDD is written in 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.)

The SDE Java Bean is generated from the XML Schema description:




*Description of Service Data Element in XML Schema*

*Java Bean*

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

The SDD is placed in an XML Schema file which will be imported into the GWSDL description of the Grid Service. We will see how to do this in the following page.

**<-- Previous ^Up^ Next -->**

*Borja Sotomayor*

# The Globus Toolkit 3 Programmer's Tutorial

## Service Data

### A service with Service Data

**<-- Previous ^Up^ Next -->**

We're 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 go to the download section.

### The MathDataType SDD

First of all, we have to define what Service Data we want to offer. We're only going to have a single SDE type, called `MathDataType`. This SDE will tell us what is the last operation done by MathService, and the total number of operations done. The following is an extract from the file `MathDataType.xsd`.

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

This is a part of $TUTORIAL_DIR/gt3tutorial/core/servicedata/schema/MathDataType.xsd

If you take a close look at `MathDataType.xsd`, 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 java.lang.String lastOp;  // attribute
    private int numOps;  // attribute

    public MathDataType() {
```

```
    }

    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;
    }
}
```

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

## Service Interface

The service interface (expressed in Java) would be the following:

```
public interface MathPortType
{
    public int add(int a, int b);

    public int subtract(int a, int b);

    public int multiply(int a, int b);

    public float divide(int a, int b);
}
```

The equivalent GWSDL file can be found at `$TUTORIAL_DIR/gt3tutorial/core/servicedata/schema/Math.gwsdl`. Unlike the previous GWSDL files, which were all very similar, this GWSDL has one important change. Remember back when we introduced GWSDL? We said that GWSDL (and WSDL 1.2) had two major improvements: PortType inheritance and *service data*. Well, now we're going to take a closer look at that second improvement.

GWSDL allows us to specify the properties of a Grid Service's service data, such as the minimum and maximum amount of SDEs of a certain type that a Grid Service can have. Let's take a look at the relevant modifications we have to do in the GWSDL description to include service data.

First of all, we need to include two new namespaces. The first corresponds to the SDE type (if you take a look at `MathDataType.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 tags.

```
<definitions name="MathService"
        targetNamespace="http://www.gt3tutorial.org/
namespaces/0.2/core/servicedata/Math"
        xmlns:tns="http://www.gt3tutorial.org/
namespaces/0.2/core/servicedata/Math"
        xmlns:data="http://www.gt3tutorial.org/
namespaces/0.2/core/servicedata/MathData"
        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 `MathDataType.xsd`.

```
<import location="MathDataType.xsd"
        namespace="http://www.gt3tutorial.org/
namespaces/0.2/core/servicedata/MathData"/>
```

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 `MathDataType.xsd` (notice how we're using the `data` namespace, which corresponds with the target namespace of `MathDataType.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, will be allowed to change during the lifetime of the Grid Service, but cannot be NULL or be modified by the client.

## Service Implementation

The're also a lot of new things in the service implementation (`$TUTORIAL_DIR/gt3tutorial/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 serviceData`) and the other is the *value* of the SDE (`MathDataType mathData`).

```
private ServiceData serviceData;
private MathDataType mathData;
```

The creation of the SDEs takes place in the `postCreate` callback method, where we will also set the initial value of the SDE.

```
public void postCreate(GridContext context) throws
GridServiceException
{
        // Create Service Data Element
        serviceData = this.getServiceDataSet().create
("MathData");

        // Set the value of the SDE to a MathDataType
instance
        mathData = new MathDataType();
        serviceData.setValue(mathData);

        // Set intial values of MathServiceData
        mathData.setLastOp("NONE");
        mathData.setNumOps(0);
```

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

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 instance's 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 inital 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.

All the public remote methods (add, subtract, multiply, and divide) now have to modify the Service Data each time they are called (to update the 'last operation' and the 'number of operations done'). We do this using the private `mathData` attribute, and an extra method called `incrementOps`.

```
public int add(int a, int b) throws RemoteException
{
        mathData.setLastOp("Addition");
        incrementOps();
        return a + b;
}
```

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

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

## Deployment Descriptor

The deployment descriptor barely changes: new name, new instance class, and new instance schema path.

```
<deployment name="defaultServerConfig" xmlns="http://xml.
apache.org/axis/wsdd/"
        xmlns:java="http://xml.apache.org/axis/wsdd/
```

```
providers/java">

        <service name="tutorial/core/servicedata/
MathFactoryService" provider="Handler" style="wrapped">
                <parameter name="name"
                        value="MathService Factory (with
Service Data)"/>
                <parameter name="instance-name"
                        value="MathService Instance (with
Service Data)"/>
                <parameter name="instance-schemaPath"
                        value="schema/gt3tutorial.core.
servicedata/Math/Math_service.wsdl"/>
                <parameter name="instance-baseClassName"
                        value="gt3tutorial.core.
servicedata.impl.MathImpl"/>

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

</deployment>
```

This file is $TUTORIAL_DIR/gt3tutorial/core/servicedata/Math.wsdd

## Namespace Mappings

We need to add the following namespace-to-package mappings to our mappings file:

```
http\://www.gt3tutorial.org/namespaces/0.2/core/
servicedata/Math=gt3tutorial.core.servicedata.wsdl
http\://www.gt3tutorial.org/namespaces/0.2/core/
servicedata/Math/bindings=gt3tutorial.core.servicedata.
wsdl.bindings
http\://www.gt3tutorial.org/namespaces/0.2/core/
```

```
servicedata/Math/service=gt3tutorial.core.servicedata.wsdl.
service
```

Add this to file $TUTORIAL_DIR/namespace2package.mappings

## Compile and deploy

Once again, we'll build the Grid Service using our handy buildfile and script:

```
./tutorial_build.sh gt3tutorial/core/servicedata/
schema/Math.gwsdl
```

Run from $TUTORIAL_DIR/

Now, deploy the GAR file, start the service container, and create a MathService instance:

```
ant deploy -Dgar.name=$TUTORIAL_DIR/build/lib/
gt3tutorial.core.servicedata.Math.gar

globus-start-container

ogsi-create-service
     http://localhost:8080/ogsa/services/tutorial/
core/servicedata/MathFactoryService
     math
```

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

Notice how we're creating one instance for all the clients (using the `ogsi-create-service` command). We can't have the clients using transient instances (i.e.having them create an instance, using it, and then destroying it), because that kind of instances are intended to avoid information sharing among clients. If we used transient instances, the service data would only be available to *one* client.

**<-- Previous ^Up^ Next -->**

*Borja Sotomayor*

# The Globus Toolkit 3 Programmer's Tutorial

## Service Data

### A client that accesses Service Data

**<-- Previous ^Up^ Next -->**

We will now modify the factory client #1 (the one that accessed a precreated instance). Instead of just adding two numbers, it will also ask for the `MathData` SDE. Before performing the addition, it will show us the content of the SDE (the previous operation, and the number of operations performed). First let's take a look at the complete source code:

```
package gt3tutorial.core.servicedata.client;

import org.gridforum.ogsi.OGSIServiceGridLocator;
import org.gridforum.ogsi.GridService;
import org.gridforum.ogsi.ExtensibilityType;
import org.gridforum.ogsi.ServiceDataValuesType;
import org.globus.ogsa.utils.AnyHelper;
import org.globus.ogsa.utils.QueryHelper;
import gt3tutorial.core.servicedata.wsdl.service.
MathServiceGridLocator;
import gt3tutorial.core.servicedata.wsdl.MathPortType;
import gt3tutorial.core.servicedata.servicedata.
MathDataType;
import java.net.URL;

public class MathClient
{
        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]);
                        int b = Integer.parseInt(args[2]);

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

                        // Get a reference to the
```

```
GridService PortType
                        OGSIServiceGridLocator locator =
new OGSIServiceGridLocator();
                        GridService gridService = locator.
getGridServicePort(GSH);

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

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

                        // Call remote method
                        int sum = math.add(a,b);

                        // Print result
                        System.out.println(a + " + " + b +
" = " + sum);
                }catch(Exception e)
                {
                        System.out.println("ERROR!");
                        e.printStackTrace();
                }
        }
}
```

This file is $TUTORIAL_DIR/gt3tutorial/core/servicedata/client/MathClient.java

The only new code (besides the different package names) is shown in bold. First of all, we need to get a reference to the GridService port type, which will allow us to access the service data (the MathService port only allows us to call add(), subtract(), etc.).

```
OGSIServiceGridLocator locator = new OGSIServiceGridLocator
();
GridService gridService = locator.getGridServicePort(GSH);
```

Now we have 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 =
        gridService.findServiceData(QueryHelper.
getNamesQuery("MathData"));
```

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

## Compile and run

Let's compile the client:

```
javac -classpath ./build/classes:$CLASSPATH
        gt3tutorial/core/servicedata/client/
MathClient.java
```

The client receives three parameters:

1. The GSH
2. First number
3. Second number

If you run it several times:

```
java gt3tutorial.core.servicedata.client.MathClient
     http://localhost:8080/ogsa/services/tutorial/
core/servicedata/MathFactoryService/math
     5
```

**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.

**<-- Previous ^Up^ Next -->**

*Borja Sotomayor*

# The Globus Toolkit 3 Programmer's Tutorial

## Service Data

### The GridService Service Data

**<-- Previous ^Up^ Next -->**

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 which, as we've seen in previous sections (for example, when seeing the Factory Client #2), is a PortType which is present in *all* Grid Services.

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 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 exhaustive 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 page).

### The `PrintGridServiceData` client

The examples file (available in the Download tutorial files page) includes a program which, given a GSH, prints all the GridService SDEs mentioned above. You can find the source code at `$TUTORIAL_DIR/gt3tutorial/core/servicedata/client/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 test the program with a Grid Service factory and a Grid Service instance, to see what Service Data is printed out. First of all, compile the client:

```
javac gt3tutorial/core/servicedata/client/
PrintGridServiceData.java
```

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.

```
java gt3tutorial.core.servicedata.client.
PrintGridServiceData
        http://localhost:8080/ogsa/services/tutorial/
core/servicedata/MathFactoryService
```

You should see the following on your terminal:

```
gridServiceHandle: http://IP_address:8080/ogsa/
services/tutorial/core/servicedata/MathFactoryService
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): Wed Jun 25 11:30:49
CEST 2003
serviceDataName: gridServiceHandle
serviceDataName: entry
serviceDataName: factoryLocator
serviceDataName: createServiceExtensibility
serviceDataName: serviceDataName
serviceDataName: interface
serviceDataName: terminationTime
serviceDataName: setServiceDataExtensibility
serviceDataName: membershipContentRule
serviceDataName: gridServiceReference
serviceDataName: findServiceDataExtensibility
interface: {http://ogsi.gridforum.org}
FactoryServiceGroup
interface: Factory
interface: ServiceGroup
interface: GridService
```

Notice how, being a factory, this Grid Service has no factory locator (`Not created by a factory!`). Also, notice how this Grid Service implements the Factory and GridService PortTypes.

Now, let's try it out with an instance. The following supposes you have created an instance called `math`. Again, you can substitute the GSH with the handle of any instance (of any Grid Service) you have created so far.

```
java gt3tutorial.core.servicedata.client.
PrintGridServiceData
        http://localhost:8080/ogsa/services/tutorial/
core/servicedata/MathFactoryService/math
```

You should see the following:

```
gridServiceHandle: http://IP_address:8080/ogsa/
services/tutorial/core/servicedata/
MathFactoryService/math
factoryLocator: http://IP_address:8080/ogsa/services/
tutorial/core/servicedata/MathFactoryService
terminationTime (after): The service plans to exist
indefinitely
terminationTime (before): The service has no plans
to terminate.
terminationTime (timestamp): Wed Jun 25 11:30:55
CEST 2003
serviceDataName: gridServiceHandle
serviceDataName: factoryLocator
serviceDataName: {http://www.gt3tutorial.org/
namespaces/0.2/core/servicedata/Math}MathData
serviceDataName: serviceDataName
serviceDataName: interface
serviceDataName: terminationTime
serviceDataName: setServiceDataExtensibility
serviceDataName: gridServiceReference
serviceDataName: findServiceDataExtensibility
interface: {http://www.gt3tutorial.org/
namespaces/0.2/core/servicedata/Math}MathPortType
interface: GridService
```

Now, notice how we can access the factory locator (since this is an instance created by a factory). Also, notice how the implemented PortTypes are GridService (the PortType all Grid Services must implement) and our very own `MathPortType`. `MathData` is also present in the `serviceDataName` SDE.

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.

**<-- Previous ^Up^ Next -->**

*Borja Sotomayor*

# The Globus Toolkit 3 Programmer's Tutorial

## Notifications

**<-- Previous ^Up^ Next -->**

In this section we will see notifications, a core service 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 two examples which use notifications. The first example is a simple one (without service data), and the second one shows how notifications are related to service data.

**<-- Previous ^Up^ Next -->**

*Borja Sotomayor*

# The Globus Toolkit 3 Programmer's Tutorial

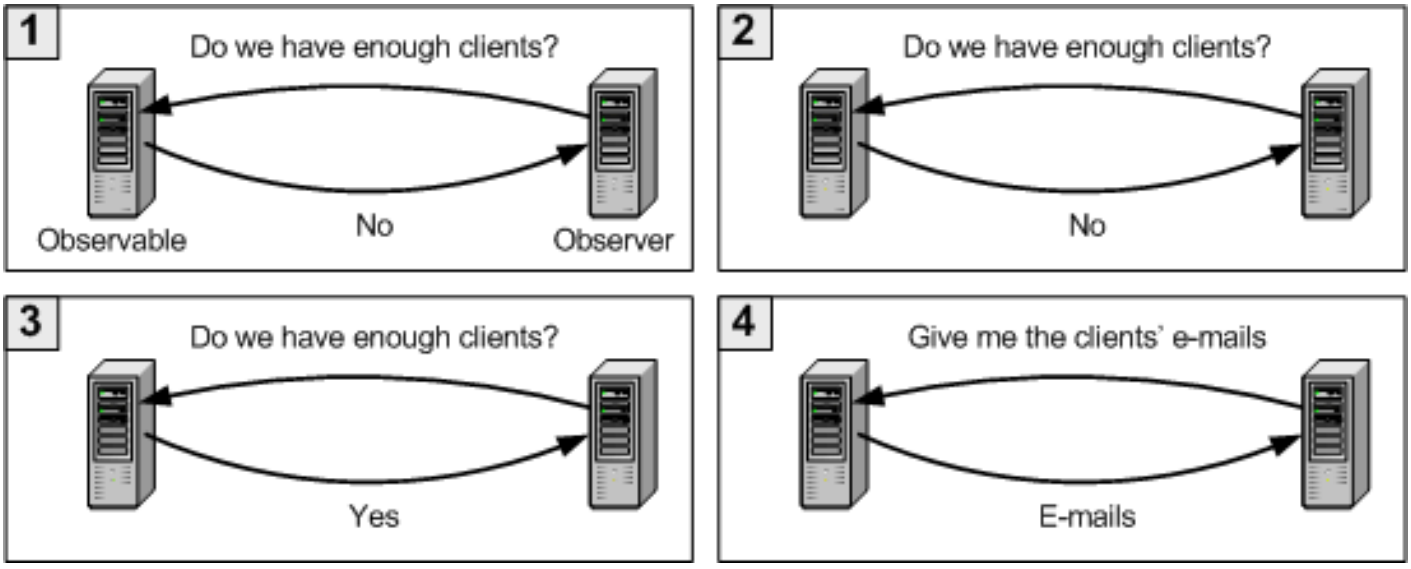
## Notifications

### What are notifications?

**<-- Previous ^Up^ Next -->**

Notifications are nothing new. It's a frequently used software design pattern, although you might know it with a different name: Observer/Observable, Model-View-Controler, 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 new value is inmediately 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:

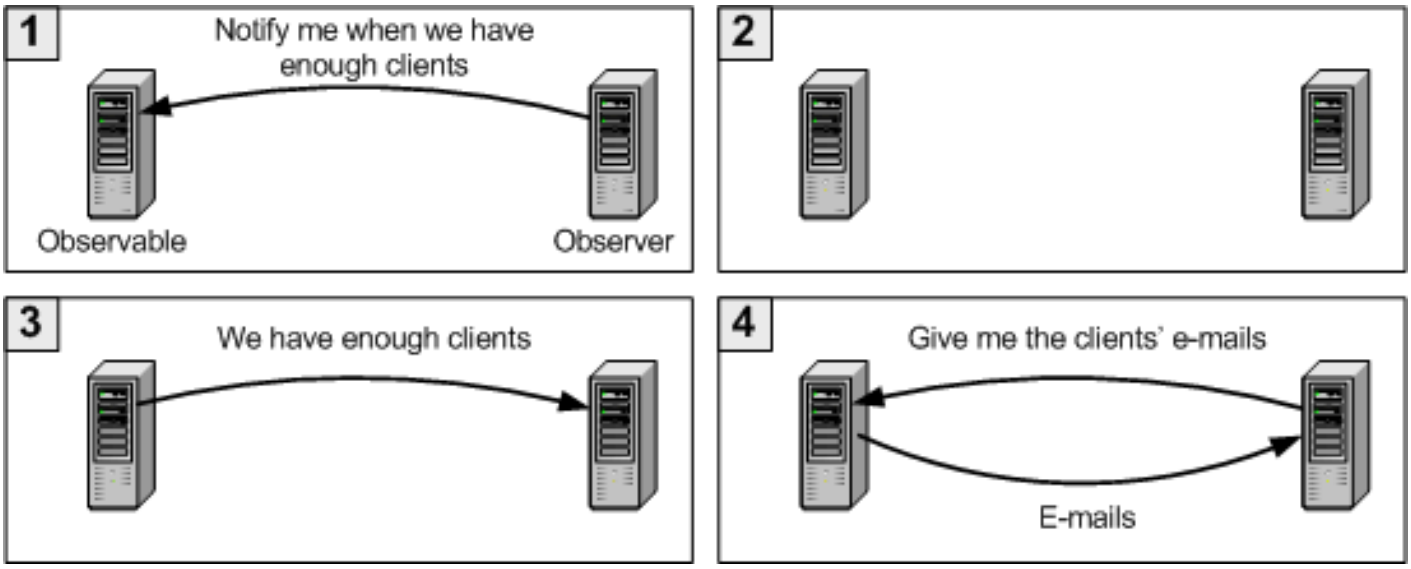



1. The observer asks the observable if there are any changes. The observable replies "No", so the observer waits a while before making another call.
2. Once again, the observer asks the observable if there are any changes. The observable replies "No", so the observer waits a while before making another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

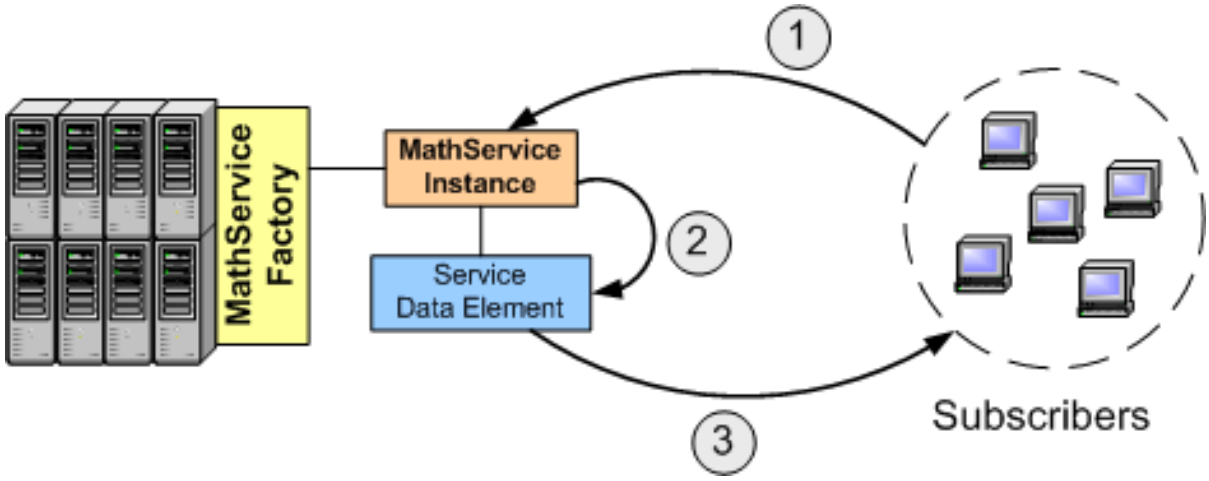
notification would include all the e-mail addresses. This approach is useful when:

- 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 loose approach limits what information the client receives, but is more efficient since we save an extra trip to the observable. In this section we'll see how to implement both pull and push notifications in GT3.

## Notifications in GT3

Notifications in GT3 are closely related to service data. In fact, the observers don't subscribe to a whole instance, but to a particular Service Data Element (SDE) in that instance. The following diagram shows how a MathService instance 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 instance 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 loose notification pattern. However, we'll see that it is also possible to implement a strict notification pattern in GT3.

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*.

**<-- Previous ^Up^ Next -->**

*Borja Sotomayor*

# The Globus Toolkit 3 Programmer's Tutorial

## Notifications

### A 'pull' notification service

**<-- Previous ^Up^ Next -->**

Let's suppose that we have several clients connected to our MathService, and that all of them want to know what the internal value is at all times. As we just saw, we could do this simply by making the clients call the `getValue` method every couple seconds. However, we're going to do this using notifications. Each time any client makes a call to `add` or `subtract`, all the subscribed clients will receive a notification.

This first example we're going to write uses a 'pull' notification pattern. This means that after a notification is received, the clients will have to make a call to `getValue` to find out what the new value is.

### Service Interface

The service interface (in Java) would be this:

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

    public void subtract(int a);

    public int getValue();
}
```

The corresponding GWSDL file can be found at `$TUTORIAL_DIR/gt3tutorial/core/notificationsPull/schema/Math.gwsdl`. It is very similar to the previous GWSDL files we've seen, with one important addition:

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

        <!-- Operations -->

</gwsdl:portType>
```

Since we want our GridService to act as a notification source, it must extend from an OGSI PortType called NotificationSource.

## Service Implementation

The service implementation has to be modified. However, in the case of a 'pull' notification pattern, these changes are very simple. First let's take a look at the whole source code, and then at the changes.

```
package gt3tutorial.core.notificationsPull.impl;

import org.globus.ogsa.impl.ogsi.GridServiceImpl;
import org.globus.ogsa.OperationProvider;
import org.globus.ogsa.GridServiceBase;
import org.globus.ogsa.GridContext;
import org.globus.ogsa.ServiceData;
import org.globus.ogsa.GridServiceCallback;
import org.globus.ogsa.GridServiceException;
import gt3tutorial.core.notificationsPull.wsdl.
MathPortType;
import java.rmi.RemoteException;
import javax.xml.namespace.QName;

public class MathProvider implements OperationProvider,
GridServiceCallback
{
        private int value = 0;
        private ServiceData serviceData;

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

        // Operation Provider methods
        public void initialize(GridServiceBase base)
throws GridServiceException
        {
                this.base = base;
        }

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

        public void postCreate(GridContext context) throws
GridServiceException
        {
                serviceData = base.getServiceDataSet().
create("DummySDE");
                base.getServiceDataSet().add(serviceData);
        }

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

```
                value = value + a;
                serviceData.notifyChange();
        }

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

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

This file has been shortened for simplicity. It is missing empty implementations of the callback methods. The complete file is `$TUTORIAL_DIR/gt3tutorial/core/notificationsPull/impl/MathProvider.java`

First of all, the service needs to have service data, so clients can subscribe to it:

```
private ServiceData serviceData;
```

The Service Data Element is created in the `postCreate` callback. Since we're implementing a 'pull' notification pattern, the SDE will be empty (notice how we're not assigning any value, like we did in the Service Data example). In a sense, this is going to be a 'dummy' SDE, which will only be used to send 'blind notifications' ("A change has occured", but without giving any more information). We'll use service data more exhaustively when we apply a 'push' notification pattern.

```
serviceData = base.getServiceDataSet().create("DummySDE");
base.getServiceDataSet().add(serviceData);
```

Finally, each time an addition or a subtraction is performed, we deliver a notification to the clients.

```
serviceData.notifyChange();
```

## Deployment Descriptor

The deployment descriptor has no new paramaters although, as usual, we have to change the `service name="..."` attribute, and the `instanceClass` and `instanceSchemaPath` parameters.

```
<?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/core/notificationsPull/
MathFactoryService" provider="Handler" style="wrapped">
                <parameter name="name"
                        value="MathService Factory (with
Notifications - Pull aproach)"/>
                <parameter name="instance-name"
                        value="MathService Instance (with
Notifications - Pull aproach)"/>
                <parameter name="instance-schemaPath"
                        value="schema/gt3tutorial.core.
notificationsPull/Math/Math_service.wsdl"/>
                <parameter name="instance-className"
                        value="gt3tutorial.core.
notificationsPull.wsdl.MathPortType" />
                <parameter name="instance-baseClassName"
                        value="org.globus.ogsa.impl.ogsi.
GridServiceImpl"/>
                <parameter name="instance-
operationProviders"
                        value="gt3tutorial.core.
notificationsPull.impl.MathProvider
                        org.globus.ogsa.impl.ogsi.
NotificationSourceProvider" />

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

Save this file as $TUTORIAL_DIR/gt3tutorial/core/notificationsPull/Math.wsdd

## Namespace Mappings

We need to add the following namespace-to-package mappings to our mappings file:

```
http\://www.gt3tutorial.org/namespaces/0.2/core/
notificationsPull/Math=gt3tutorial.core.notificationsPull.
wsdl
http\://www.gt3tutorial.org/namespaces/0.2/core/
notificationsPull/Math/bindings=gt3tutorial.core.
notificationsPull.wsdl.bindings
http\://www.gt3tutorial.org/namespaces/0.2/core/
notificationsPull/Math/service=gt3tutorial.core.
notificationsPull.wsdl.service
```

Add this to file $TUTORIAL_DIR/namespace2package.mappings

## Compile and deploy

Once again, we'll build the Grid Service using our handy buildfile and script:

```
./tutorial_build.sh gt3tutorial/core/
notificationsPull/schema/Math.gwsdl
```

Run from $TUTORIAL_DIR/

Now, deploy the GAR file, start the service container, and create a MathService instance:

```
ant deploy -Dgar.name=$TUTORIAL_DIR/build/lib/
gt3tutorial.core.notificationsPull.Math.gar

globus-start-container

ogsi-create-service
     http://localhost:8080/ogsa/services/tutorial/
core/notificationsPull/MathFactoryService
     math
```

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

It is important to realize that, in this scenario, we need to create an instance which clients will subscribe to. We can't have the clients using transient instances (i.e.having them create an instance, using it, and then destroying it), because that kind of instances are intended to avoid information sharing among clients. This doesn't necesarily mean that we can't use notifications with transient services: a client might use an instance for several hours and, despite being the only subscriber, he might need to receive

notifications (e.g.changes in the Grid Service which are done by a third system, which is not a Grid Service client).

**<-- Previous ^Up^ Next -->**

*Borja Sotomayor*

# The Globus Toolkit 3 Programmer's Tutorial

## Notifications

### A 'pull' notification client

**<-- Previous ^Up^ Next -->**

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 (of more correctly: a class which is going to receive notification) *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'.

### Listener Client

```
package gt3tutorial.core.notificationsPull.client;

import org.globus.ogsa.client.managers.
NotificationSinkManager;
import org.globus.ogsa.NotificationSinkCallback;
import org.globus.ogsa.impl.core.service.
ServicePropertiesImpl;
import org.gridforum.ogsi.ExtensibilityType;
import org.gridforum.ogsi.HandleType;
import gt3tutorial.core.notificationsPull.wsdl.service.
MathServiceGridLocator;
import gt3tutorial.core.notificationsPull.wsdl.
MathPortType;
import java.net.URL;
import java.rmi.RemoteException;

public class MathListener extends ServicePropertiesImpl
implements NotificationSinkCallback
{
        private MathPortType math;

        public static void main(String[] args)
        {
                try
                {
                        // Get command-line arguments
```

```
                        HandleType GSH = new HandleType
(args[0]);
                        MathListener mathListener = new
MathListener(GSH);
                }catch(Exception e)
                {
                        System.out.println("ERROR!");
                        e.printStackTrace();
                }
        }

        public MathListener(HandleType GSH) throws
Exception
        {
                // Get a reference to the Grid Service
instance
                MathServiceGridLocator mathServiceLocator
= new MathServiceGridLocator();
                math = mathServiceLocator.
getMathServicePort(GSH);

                // Start listening to the MathService
                NotificationSinkManager notifManager =
NotificationSinkManager.getManager();
                notifManager.startListening
(NotificationSinkManager.MAIN_THREAD);
                String sink  =  notifManager.addListener
("DummySDE", 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
                {
                        // Value has changed, so we need
to get the new value
                        int value = math.getValue();
                        System.out.println("Value has
changed: " + value);
                }catch(Exception exc)
                {
                        System.out.println("ERROR!");
                        exc.printStackTrace();
                }
        }
}
```

Save this file as $TUTORIAL_DIR/gt3tutorial/core/notificationsPull/client/MathListener.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 (`getValue`), 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 gets the only argument to the program (the MathService instance URL) 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("DummySDE", 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 atention. 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. However, since this is a 'pull' notification pattern, we're not going to use it. When a notification arrives, we make a call to `getValue` and print the current value on screen.

Finally, let's compile and run the client:

```
javac -classpath ./build/classes:$CLASSPATH
        gt3tutorial/core/notificationsPull/client/
MathListener.java

java -Dorg.globus.ogsa.schema.root=http://
localhost:8080/
      gt3tutorial.notificationsStrict.client.
MathListener
      http://localhost:8080/ogsa/services/tutorial/
core/notificationsPull/MathFactoryService/math
```

Read this FAQ entry if you get an error related to the `client-server-config.wsdd` file.

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. That's what the adder client will take care of.

## Adder Client

The code for this client is pretty straightfoward, and is identical to previous examples.

```
package gt3tutorial.core.notificationsPull.client;

import gt3tutorial.core.notificationsPull.wsdl.service.
MathServiceGridLocator;
import gt3tutorial.core.notificationsPull.wsdl.
MathPortType;
import java.net.URL;

public class MathAdder
{
        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();
                }
    }
}
```

Save this file as $TUTORIAL_DIR/gt3tutorial/core/notificationsPull/client/MathAdder.java

Now, compile and run it:

```
javac -classpath ./build/classes:$CLASSPATH
        gt3tutorial/core/notificationsPull/client/
MathAdder.java

java gt3tutorial.core.notificationsPull.client.
MathAdder
     http://localhost:8080/ogsa/services/tutorial/
core/notificationsPull/MathFactoryService/math
     15
```

Remember to keep the listener client running somewhere else (another terminal, etc.) Once you run the adder client, the listener should receive a notification, informing you of the new value. To see the full potential of notification, try opening up several listeners, and then running the adder.

**<-- Previous ^Up^ Next -->**

*Borja Sotomayor*

# The Globus Toolkit 3 Programmer's Tutorial

## Notifications

### A 'push' notification service

**<-- Previous ^Up^ Next -->**

Now we're going to write up the same example, except we're going to use a 'push' notification pattern. Instead of making an extra call to MathService each time we get a notification, the information we need (the internal value of MathService) will be sent along with the notification. This example is very similar to the one seen in the Service Data section, so you might want to review it first.

### The MathDataType SDD

Since we need to send the internal value with the notification, the MathService SDE can't be empty like in the previous example. It will contain the value of the additions and subtractions performed by clients, along with other information. The SDD would look like this:

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

This is an extract from file $TUTORIAL_DIR/gt3tutorial/core/notificationsPush/schema/MathDataType.xsd

### Service Interface

The service interface is almost the same. However, notice how the `getValue()` method has disappeared. For this example we won't need it any more, since the value will travel along with the notification. The Java interface would be:

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

    public void subtract(int a);
}
```

The corresponding GWSDL file can be found at `$TUTORIAL_DIR/gt3tutorial/core/notificationsPush/schema/Math.gwsdl`. It is very similar to the GWSDL file we saw in the Service Data section. As in that GWSDL file, we need to include the namespace of the SDD:

```
<definitions name="MathService"
    targetNamespace="http://www.gt3tutorial.org/
namespaces/0.2/core/notificationsPush/Math"
    xmlns:tns="http://www.gt3tutorial.org/namespaces/0.2/
core/notificationsPush/Math"
    xmlns:data="http://www.gt3tutorial.org/namespaces/0.2/
core/notificationsPush/MathData"
    xmlns:ogsi="http://www.gridforum.org/
namespaces/2003/03/OGSI"
    xmlns:gwsdl="http://www.gridforum.org/
namespaces/2003/03/gridWSDLExtensions"
    xmlns:sd="http://www.gridforum.org/namespaces/2003/03/
serviceData"
    xmlns:xsd="http://www.w3.org/2001/XMLSchema"
    xmlns="http://schemas.xmlsoap.org/wsdl/">
```

Import the XML Schema file that contains the SDD:

```
<import location="MathDataType.xsd"
    namespace="http://www.gt3tutorial.org/namespaces/0.2/
core/notificationsPush/MathData"/>
```

And, finally, extend the NotificationSource PortType and specify the characteristics of the service data element of our Grid Service (which, as in the Service Data section, will be called `MathData`).

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

        <!-- <operation>s -->

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

</gwsdl:portType>
```

## Service Implementation

The service implementation has to be modified, and now looks very similar to the Service Data

example we saw earlier. Let's take a look at the whole source code, and then at the changes.

```
package gt3tutorial.core.notificationsPush.impl;

import org.globus.ogsa.impl.ogsi.GridServiceImpl;
import org.globus.ogsa.OperationProvider;
import org.globus.ogsa.GridServiceBase;
import org.globus.ogsa.GridContext;
import org.globus.ogsa.ServiceData;
import org.globus.ogsa.GridServiceCallback;
import org.globus.ogsa.GridServiceException;
import gt3tutorial.core.notificationsPush.wsdl.
MathPortType;
import gt3tutorial.core.notificationsPush.servicedata.
MathDataType;
import java.rmi.RemoteException;
import javax.xml.namespace.QName;

public class MathProvider implements OperationProvider,
GridServiceCallback
{
        private ServiceData serviceData;
        private MathDataType mathData;

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

        // Operation Provider methods
        public void initialize(GridServiceBase base)
throws GridServiceException
        {
                this.base = base;
        }

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

        public void postCreate(GridContext context) throws
GridServiceException
        {
                serviceData = base.getServiceDataSet().
create("MathData");

                mathData = new MathDataType();
                serviceData.setValue(mathData);

                mathData.setValue(0);
                mathData.setLastOp("NONE");
                mathData.setNumOps(0);

                base.getServiceDataSet().add(serviceData);
        }
```

```
        private void incrementOps()
        {
                int numOps = mathData.getNumOps();
                mathData.setNumOps(numOps + 1);
        }

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

        public void subtract(int a) throws RemoteException
        {
                mathData.setLastOp("Subtraction");
                incrementOps();
                mathData.setValue(mathData.getValue() - a);
                serviceData.notifyChange();
        }
}
```

Save this file as $TUTORIAL_DIR/gt3tutorial/core/notificationsPush/impl/MathProvider.java

First of all, we no longer have a `private int value` attribute, since that information is now contained in the MathDataType value of the MathData SDE:

```
private MathDataType mathData;
```

The SDE is no longer a 'dummy SDE' (like the one we used in the 'pull' notification pattern). It will contain real data, so we have to create a `MathDataType` instance and give it initial values. We do so in the PostCreate callback method.

```
mathData = new MathDataType();
serviceData.setValue(mathData);

mathData.setValue(0);
mathData.setLastOp("NONE");
mathData.setNumOps(0);
```

Both the addition and subtraction operations have changed. Instead of working with the `value` attribute, we now work with the MathData SDE. The new value is set there, along with the 'previous operation' and the 'number of operations'.

```
mathData.setLastOp("Addition");
incrementOps();
mathData.setValue(mathData.getValue() + a);
```

```
serviceData.notifyChange();
```

The number of operations is increased in a private method called `incrementOps()`:

```
private void incrementOps()
{
        int numOps = mathData.getNumOps();
        mathData.setNumOps(numOps + 1);
}
```

## Deployment Descriptor

The deployment descriptor remains practically the same:

```
<?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/core/notificationsPush/
MathFactoryService" provider="Handler" style="wrapped">
                <parameter name="name"
                        value="MathService Factory (with
Notifications - Push aproach)"/>
                <parameter name="instance-name"
                        value="MathService Instance (with
Notifications - Push aproach)"/>
                <parameter name="instance-schemaPath"
                        value="schema/gt3tutorial.core.
notificationsPush/Math/Math_service.wsdl"/>
                <parameter name="instance-className"
                        value="gt3tutorial.core.
notificationsPush.wsdl.MathPortType" />
                <parameter name="instance-baseClassName"
                        value="org.globus.ogsa.impl.ogsi.
GridServiceImpl"/>
                <parameter name="instance-
operationProviders"
                        value="gt3tutorial.core.
notificationsPush.impl.MathProvider
                        org.globus.ogsa.impl.ogsi.
NotificationSourceProvider" />

                <!-- Start common parameters -->
                <parameter name="allowedMethods" value="*"/
>
                <parameter name="persistent" value="true"/>
                <parameter name="className"
                        value="org.gridforum.ogsi.
NotificationFactory"/>
```

```
                <parameter name="baseClassName"
                        value="org.globus.ogsa.impl.ogsi.
PersistentGridServiceImpl"/>
                <parameter name="schemaPath"
                        value="schema/ogsi/
ogsi_notification_factory_service.wsdl"/>
                <parameter name="handlerClass"
                        value="org.globus.ogsa.handlers.
RPCURIProvider"/>
                <parameter name="factoryCallback"
                        value="org.globus.ogsa.impl.ogsi.
DynamicFactoryCallbackImpl"/>
                <parameter name="operationProviders"
                        value="org.globus.ogsa.impl.ogsi.
FactoryProvider
                        org.globus.ogsa.impl.ogsi.
NotificationSourceProvider"/>
        </service>

</deployment>
```

Save this file as $TUTORIAL_DIR/gt3tutorial/core/notificationsPush/Math.wsdd

## Namespace Mappings

We need to add the following namespace-to-package mappings to our mappings file:

```
http\://www.gt3tutorial.org/namespaces/0.2/core/
notificationsPush/Math=gt3tutorial.core.notificationsPush.
wsdl
http\://www.gt3tutorial.org/namespaces/0.2/core/
notificationsPush/Math/bindings=gt3tutorial.core.
notificationsPush.wsdl.bindings
http\://www.gt3tutorial.org/namespaces/0.2/core/
notificationsPush/Math/service=gt3tutorial.core.
notificationsPush.wsdl.service
```

Add this to file $TUTORIAL_DIR/namespace2package.mappings

## Compile and deploy

Once again, we'll build the Grid Service using our handy buildfile and script:

```
./tutorial_build.sh gt3tutorial/core/
notificationsPush/schema/Math.gwsdl
```

Run from $TUTORIAL_DIR/

Now, deploy the GAR file, start the service container, and create a MathService instance:

```
ant deploy -Dgar.name=$TUTORIAL_DIR/build/lib/
gt3tutorial.core.notificationsPush.Math.gar

globus-start-container

ogsi-create-service
     http://localhost:8080/ogsa/services/tutorial/
core/notificationsPush/MathFactoryService
     math
```

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

As in the previous example, we're creating an instance ourselves, instead of letting the clients create and destroy their own instances. This way, we'll be able to subscribe multiple clients to the same MathService instance and see the notifications in action.

**<-- Previous  ^Up^  Next -->**

*Borja Sotomayor*

# The Globus Toolkit 3 Programmer's Tutorial

## Notifications

### A 'push' notification client

**<-- Previous ^Up^**

### Math Listener

The code for the client is almost identical to the 'pull' notification client.

```
package gt3tutorial.core.notificationsPush.client;

import org.globus.ogsa.client.managers.
NotificationSinkManager;
import org.globus.ogsa.NotificationSinkCallback;
import org.globus.ogsa.impl.core.service.
ServicePropertiesImpl;
import org.gridforum.ogsi.ExtensibilityType;
import org.gridforum.ogsi.HandleType;
import org.gridforum.ogsi.ServiceDataValuesType;
import org.globus.ogsa.utils.AnyHelper;
import gt3tutorial.core.notificationsPull.wsdl.service.
MathServiceGridLocator;
import gt3tutorial.core.notificationsPull.wsdl.
MathPortType;
import gt3tutorial.core.notificationsPush.servicedata.
MathDataType;
import java.net.URL;
import java.rmi.RemoteException;

public class MathListener extends ServicePropertiesImpl
implements NotificationSinkCallback
{
        private MathPortType math;

        public static void main(String[] args)
        {
                try
                {
                        // Get command-line arguments
                        HandleType GSH = new HandleType
(args[0]);
                        MathListener mathListener = new
MathListener(GSH);
                }catch(Exception e)
```

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

        public MathListener(HandleType GSH) throws
Exception
        {
                // Get a reference to the Grid Service
instance
                MathServiceGridLocator mathServiceLocator
= new MathServiceGridLocator();
                math = mathServiceLocator.
getMathServicePort(GSH);

                // 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(serviceData,
MathDataType.class);

                        // 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();
                }
        }
}
```

Save this file as $TUTORIAL_DIR/gt3tutorial/core/notificationsPush/client/MathListener.java

The main changes are in the `deliverNotification` method. Now, instead of calling the `getValue` method, we access the SDE which had been sent along with the notification and that already includes the information we want to show on screen. As we saw in the Service Data section, we first have to cast the `ExtensibilityTYpe` into a `MathDataType` using 'helper' classes.

Also, since we don't have to make any calls to MathService, we don't have to get a reference to a MathPortType like we did in the previous example.

Now, let's compile and run the client:

```
javac -classpath ./build/classes:$CLASSPATH
        gt3tutorial/core/notificationsPush/client/
MathListener.java

java -Dorg.globus.ogsa.schema.root=http://
localhost:8080/
      gt3tutorial.core.notificationsPush.client.
MathListener
      http://localhost:8080/ogsa/services/tutorial/
core/notificationsPush/MathFactoryService/math
```

Read this FAQ entry if you get an error related to the `client-server-config.wsdd` file.

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 practically the same:

```
package gt3tutorial.core.notificationsPush.client;

import gt3tutorial.core.notificationsPush.wsdl.service.
MathServiceGridLocator;
import gt3tutorial.core.notificationsPush.wsdl.
MathPortType;
import java.net.URL;

public class MathAdder
```

```
{
        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();
                }
    }
}
```

Save this file as $TUTORIAL_DIR/gt3tutorial/core/notificationsPush/client/MathAdder.java

Compile and run it:

```
javac -classpath ./build/classes:$CLASSPATH
        gt3tutorial/notificationsPush/client/
MathAdder.java

java gt3tutorial.core.notificationsPush.client.
MathAdder
     http://localhost:8080/ogsa/services/tutorial/
core/notificationsPush/MathFactoryService/math
     30
```

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'.

**<-- Previous  ^Up^**

*Borja Sotomayor*

# The Globus Toolkit 3 Programmer's Tutorial

## Frequently Asked Questions

**<-- Back**

**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 (in the Globus website).



### I get a `NoClassDefFoundError` when trying to run a GT3 command.

Whenever you run a GT3 command without `Ant` (for example, to create a service instance), you need to manually set a couple of environment variables first. You can do this by going to your GT3 installation root, and running the following command:

```
. ./setenv.sh
```

You *must* run this from your GT3 installation root. Running from any other directory will not work. Also, notice the single dot before `./setenv.sh`. That is not a typo, you really have to write that.

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).

### I get a lot of `cannot resolve symbol` errors when trying to compile a client (mostly in Globus classes)

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` script takes care of it). 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 (versions of the tutorial previous to 0.2.1 did not include this argument).

## I followed all the steps in "Writing your first Grid Service" to write a different Grid Service, but I can't get it to work.

The MathService example is compiled and deployed step by step simply so you can get acquainted with all the steps involved in writing a Grid Service. The particular steps described in Writing your first Grid Service work for MathService, but probably won't work for other Grid Services (without major changes). You should *always* use `Ant` for 'real' Grid Services. Also, remember the first MathService isn't even a Grid Service: it's a plain vanilla Web Service deployed in the GT3 container. Bottom line: always use `Ant`.

## Can I use the 'handy multipurpose' buildfile for personal projects? If so, what limitations does it have?

Feel free to use the handy buildfile and script 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 (version 0.4 of the tutorial? :-) I hope to thoroughly document and expand the 'handy multipurpose' buildfile so it can be easily used in non-tutorial examples.

## I get a weird error when trying to run the `MathListener` program in the notification examples. Something about not finding the `client-server-config.wsdd` file. What's the problem?

This is actually a bug in the final release of GT3, which will be fixed in future releases. If you're encountering this error, you can try *one* of the following workarounds:

- Put $GLOBUS_LOCATION in your $CLASSPATH.
- Copy the `client-server-config.wsdd` file from $GLOBUS_LOCATION to the directory where `MathListener.class` is located.
- Run the listener from $GLOBUS_LOCATION.

## Is the GT3 Programmer's Tutorial available for download as a single file? (something easy to print)

The GT3 Programmer's Tutorial is currently only available as multiple HTML files. However, I plan to eventually change to an SGML or XML format (such as DocBook) so other formats (besides the

current 'multiple HTML files' format) can be generated. This would include a single downloadable PDF or HTML file.

**<-- Back**

*Borja Sotomayor*

# The Globus Toolkit 3 Programmer's Tutorial

## How to...

### ...write a GWSDL description of your Grid Service

**<-- Back**

This "How to..." page 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 primive 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.gt3tutorial.org/
namespaces/0.2/core/gwsdl/Math"
        xmlns:tns="http://www.gt3tutorial.org/
namespaces/0.2/core/gwsdl/Math"
        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 (similar. 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). Furthermore, notice how we can make up the target namespace (the `gt3tutorial.org` doesn't actually exist (an XML Namespace has to be a URI, but doesn't have to actually point anywhere).

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 section on writing Grid Services with GWSDL, 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 succesfully (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>

<!-- 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.gt3tutorial.org/
namespaces/0.2/core/gwsdl/Math"
        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>

<!-- Messages -->

<!-- PortType -->

</definitions>
```

The <types> tag contains an <xsd:schema> tag. The attributes of the <xsd:schema> should be copied verbatim, except the targetNamespace, which should be the same as the target namespace of the 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.gt3tutorial.org/
namespaces/0.2/core/gwsdl/Math"
        xmlns:tns="http://www.gt3tutorial.org/
namespaces/0.2/core/gwsdl/Math"
        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.gt3tutorial.org/
namespaces/0.2/core/gwsdl/Math"
        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.

**<-- Back**

*Borja Sotomayor*

# The Globus Toolkit 3 Programmer's Tutorial

## How to...

### ...setup the GT3 command line clients (`globus-start-container`, `ogsi-create-service`, ...)

**<-- Back**

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
        math
```

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
     math
```

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_DIRECTORY` 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 `$GLOBUS_DIRECTORY/bin`. You can now run the commands from that directory.
4. (Optional) To run the commands from any directory, add `$GLOBUS_DIRECTORY/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_DIRECTORY/docs/users_guide.html` (11 - Command Line Clients)

**<-- Back**

*Borja Sotomayor*

# The Globus Toolkit 3 Programmer's Tutorial

## Appendix

### Stub security options

**<-- Back**

### Authentication

```
stub._setProperty(Constants.GSI_SEC_CONV,Constants.
ENCRYPTION);
```

```
stub._setProperty(Constants.GSI_SEC_CONV,Constants.
SIGNATURE);
```

### X.509 Signature

```
stub._setProperty(Constants.GSI_XML_SIGNATURE,Boolean.
TRUE);
```

```
stub._setProperty(Constants.GSI_XML_SIGNATURE,Boolean.
FALSE);
```

### Authorization

```
stub._setProperty(Constants.AUTHORIZATION,NoAuthorization.
getInstance());
```

```
stub._setProperty(Constants.AUTHORIZATION,
SelfAuthorization.getInstance());
```

```
stub._setProperty(Constants.AUTHORIZATION,
HostAuthorization.getInstance());
```

## Delegation

```
stub._setProperty(GSIConstants.GSI_MODE,GSIConstants.
GSI_MODE_NO_DELEG);
```

```
stub._setProperty(GSIConstants.GSI_MODE,GSIConstants.
GSI_MODE_LIMITED_DELEG);
```

```
stub._setProperty(GSIConstants.GSI_MODE,GSIConstants.
GSI_MODE_FULL_DELEG);
```

**<-- Back**

*Borja Sotomayor*

# The Globus Toolkit 3 Programmer's Tutorial

## Introduction

**^Up^ Next -->**

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 2 main areas:

- **Introduction**: Includes information about the whole tutorial, as well as 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.

Future versions of the tutorial will include a third main area related to GT3 Higher-Level Services (programming Grid Services which use GT3 services such as Security, Job Management, File Transfer, etc.)

**^Up^ Next -->**

*Borja Sotomayor*

# The Globus Toolkit 3 Programmer's Tutorial

## Introduction

### Download tutorial files

**<-- Previous ^Up^ Next -->**

The following files are available for download:

- Tutorial examples. All the source files for the examples. The directory structure is the same as the one described in the tutorial.
- Handy multipurpose buildfile. This is the buildfile described in the Our handy multipurpose buildfile section.
- Handy multipurpose shell script. This is the script described in the Our handy multipurpose buildfile section.

**<-- Previous ^Up^ Next -->**

*Borja Sotomayor*

# The Globus Toolkit 3 Programmer's Tutorial

## Introduction

### About the author

**<-- Previous ^Up^ Next -->**

The Globus Toolkit 3 Programmer's Tutorial is maintained by Borja Sotomayor, a Computer Science Engineering student in the University of Deusto (Bilbao, Spain). You can find out more about me in my weblog, BorjaNet.

### Ackowledgements

This tutorial can hardly be considered a one-person effort. A lot of people have contributed to this tutorial with their insightful comments and their support:

- Lisa Childers
- Rebeca Cortazar
- Fernando Fraticelli (for his excellent bug hunting skills! :-)
- Leon Kuntz
- Thomas Sandholm
- Michael Schneider

**<-- Previous ^Up^ Next -->**

*Borja Sotomayor*