A short introduction to Web Services

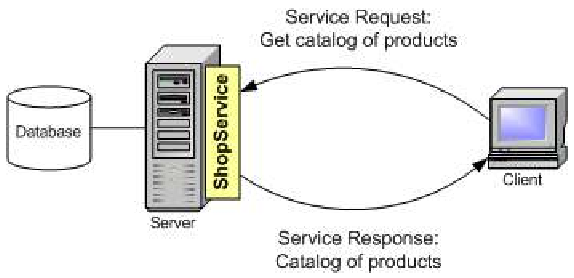
Since Web Services are the basis for Grid Services, understanding the Web Services architecture is fundamental to using GT3 and programming Grid Services.

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

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

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

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



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

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

Of course, Web Services also have some disadvantages:

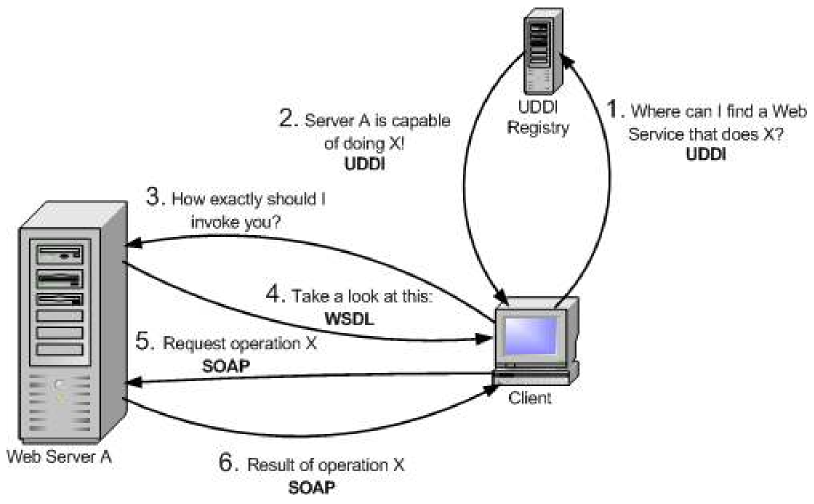
- Overhead. Transmitting all your data in XML is obviously not as efficient as using a proprietary binary code. What you win in portability, you lose in efficiency. Even so, this overhead is usually acceptable for most applications, but you will probably never find a critical real-time application that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

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

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

A Typical Web Service Invocation

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


UDDI Registry
1. Where can I find a Web Service that does X? UDDI
2. Server A is capable of doing X! UDDI
3. How exactly should I invoke you?
4. Take a look at this: WSDL
5. Request operation X SOAP
6. Result of operation X SOAP
Web Server A
Client

1. As we said before, a client may have no knowledge of what Web Service it is going to invoke. So, our first step will be to *find* a Web Service that meets our requirements. For example, we might be interested in locating a public Web Service which can give me the temperature in US cities. We'll do this by contacting a UDDI registry.

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

3. We now know the location of a Web Service, but we have no idea of how to actually invoke it. Sure, we know it can give me the temperature of a US city, buy what is the actual service invocation? The method I have to invoke might be called Temperature getCityTemperature(int CityPostalCode), but it could also be called int getUSCityTemp(string cityName, bool isFarenheit). We have to ask the Web Service to *describe* itself (i.e. tell us how exactly we should invoke it)

4. The Web Service replies in a language called WSDL.

5. We finally know where the Web Service is located and how to invoke it. The invocation itself is done in a language called SOAP. Therefore, we will first send a *SOAP request* asking for the temperature of a certain city.

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

Web Services Addressing

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

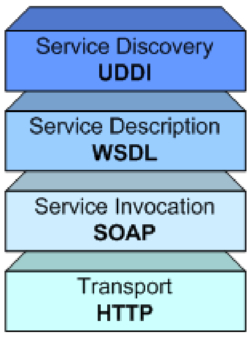
For example, the UDDI registry might have replied with the following URI:

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

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

Web Services Architecture

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



- **Service Discovery:** This part of the architecture allows us to find Web Services which meet certain requirements. This part is usually handled by UDDI (Universal Description, Discovery, and 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 responses. In theory, we could use other service invocation languages (such as XML-RPC, or even some *ad hoc* XML language). However, SOAP is by far the most popular choice for Web Services.
- **Transport** : Finally, all these messages must be transmitted somehow between the server and the client. The protocol of choice for this part of the architecture is HTTP (HyperText Transfer Protocol), the same protocol used to access conventional web pages on the Internet. Again, in theory we could be able to use other protocols, but HTTP is currently the most used one.

What a Web Service Application Looks Like

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

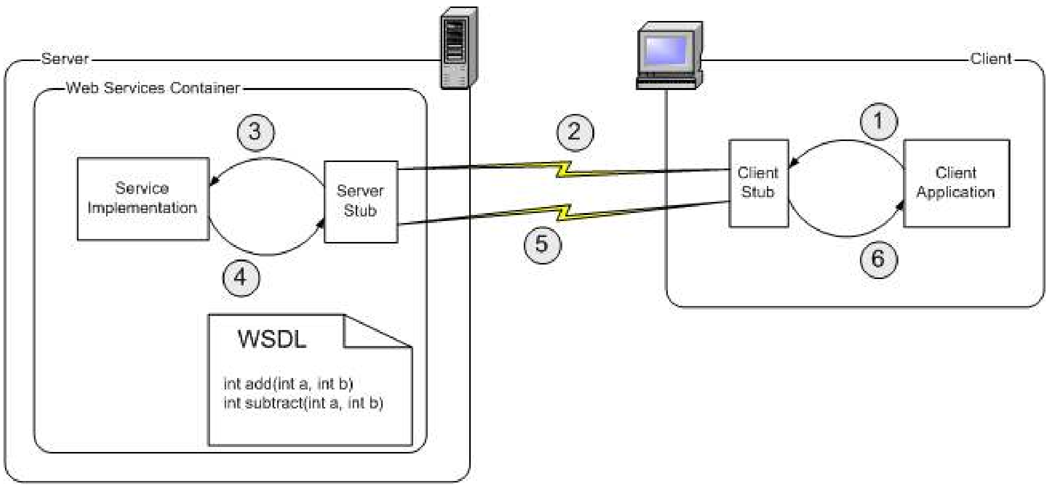
First of all, you should know that despite having a lot of protocols and languages floating around, Web Services programmers usually never write a single line of SOAP or WSDL. Once we've reached a point where our client application needs to invoke a Web Service, we *delegate* that task on a piece of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

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

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

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

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



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

1. Whenever the client application needs to invoke the Web Service, it will actually call the client stub. The client stub will turn this 'local invocation' into a proper SOAP request. This is often called the *marshaling* or *serializing* process.
2. The SOAP request is sent over a network using the HTTP protocol. The Web Services container receives the SOAP requests and hands it to the server stub. The server stub will convert the SOAP request into something the service implementation can understand (this is usually called *unmarshaling* or *deserializing*)
3. The service implementation receives the request from the service stub, and carries out the work it has been asked to do. For example, if we are invoking the int add(int a, int b) method, the service implementation will perform an addition.
4. The result of the requested operation is handed to the server stub, which will turn it into a SOAP response.
5. The SOAP response is sent over a network using the HTTP protocol. The client stub receives the SOAP response and turns it into something the client application can understand.
6. Finally the application receives the result of the Web Service invocation and uses it.

By the way, in case you're wondering, most of the Web Services Architecture is specified and standardized by the World Wide Web Consortium, the same organization responsible for XML, HTML, CSS, etc.

Prev Up Next
OGSA, OGSI, and GT3 Home What is a Grid Service?