WEB SERVICES

Shruti Gupta

June, 2014

Complete Guide

Advantages

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).

Disadvantage

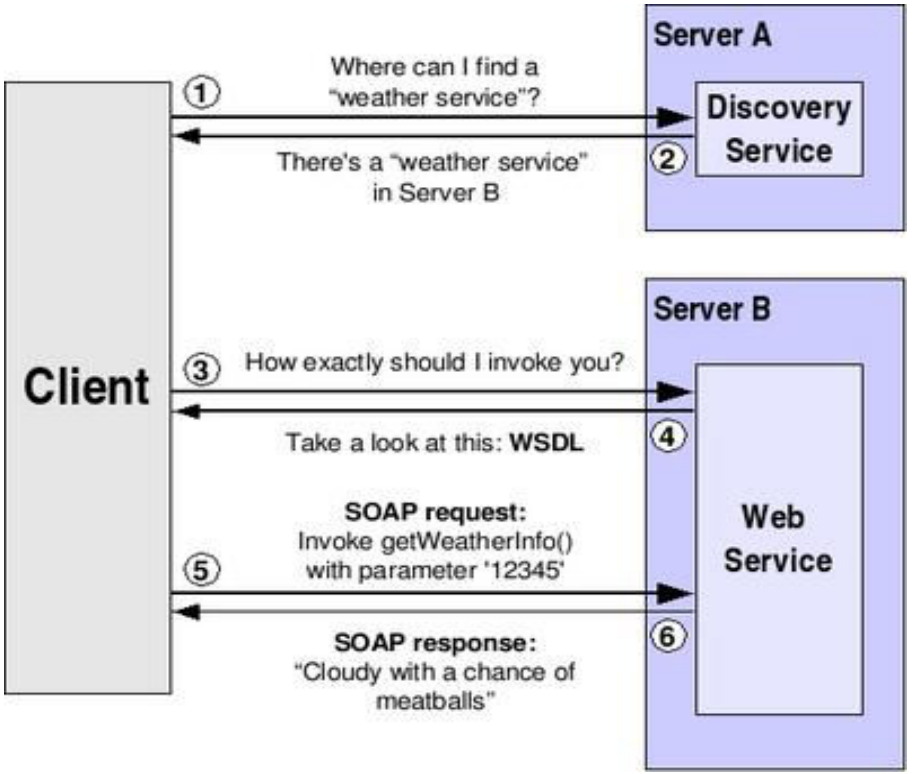
Of course, Web Services also have some disadvantages:

- 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.*

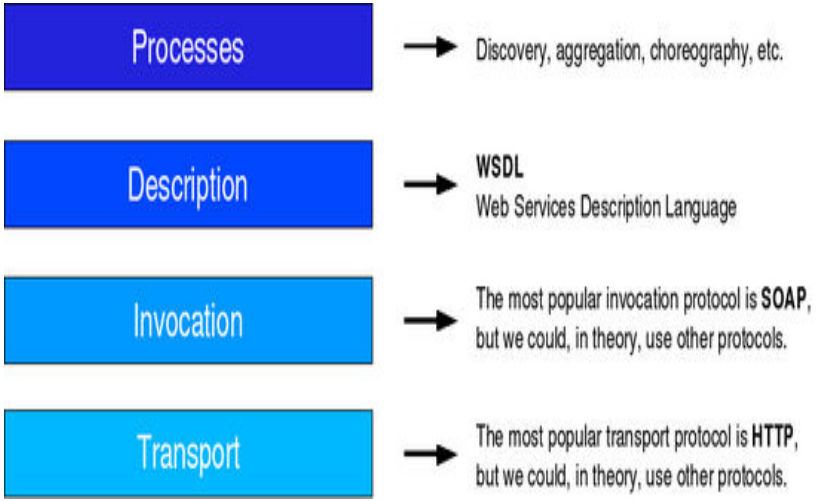
A Typical Web Service invocation

High level overview..



Web Service Architecture

High Level Design..



Architecture Description

- **Service Processes**: Discovery belongs in this part of the architecture, since it allows us to locate one particular service from among a collection of Web services.
- **Service Description**: 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.

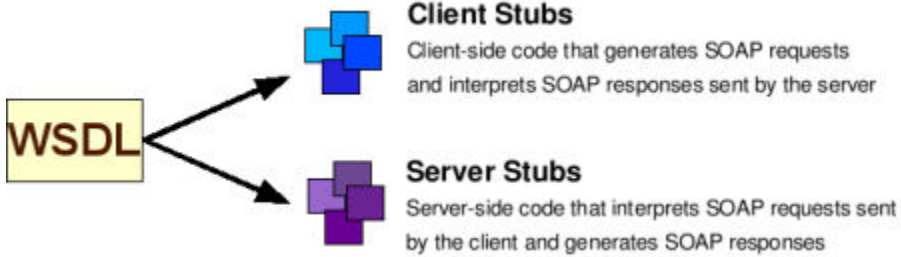
Web Service Addressing

- Addressed as URIs (Uniform Resource Identifier) rather than URL. You cannot type it in the regular address bar of the browser, it will return an error.

- URIs are supposed to be fed by a client (software) application, rather than manually by humans.

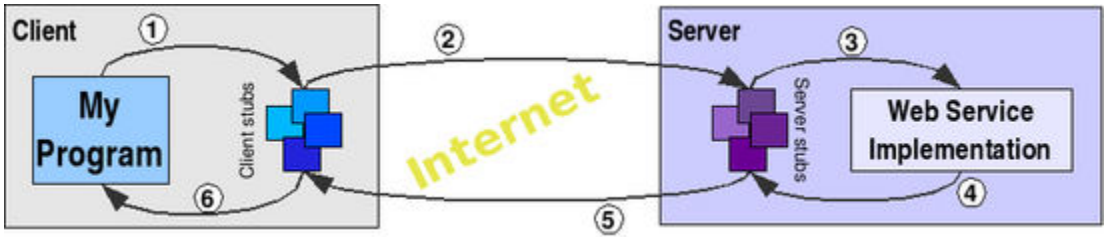
Web Service Invocation (detailed..)

- Once we've reached a point where our client application needs to invoke a Web Service, we *delegate* that task on a piece of software called a *stub*. The good news is that there are plenty of tools available that will generate stubs automatically for us, usually based on the WSDL description of the Web Service.



Web Service Invocation (detailed..)

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



Web Service Invocation (detailed..)

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

Server Side Design..

- **Web service**: First and foremost, we have our Web service. As we have seen, this is basically a piece of software that exposes a set of operations. For example, if we are implementing our Web service in Java, our service will be a Java class (and the operations will be implemented as Java methods). Obviously, we want a set of clients to be able to invoke those operations. However, our Web service implementation knows nothing about how to interpret SOAP requests and how to create SOAP responses. That's why we need a...
- **SOAP engine**: This is a piece of software that knows how to handle SOAP requests and responses. In practice, it is more common to use a generic SOAP engine than to actually generate server stubs for each individual Web service (note, however, that we still need client stubs for the client). One good example of a SOAP engine is Apache Axis. However, the functionality of the SOAP engine is usually limited to manipulating SOAP. To actually function as a server that can receive requests from different clients, the SOAP engine usually runs within an...**Application server**