Scientific Applications as Web Services: A Simple Guide

Marlon Pierce
Geoffrey Fox

Indiana University
September 8 2003

Introduction

We have discussed in several columns Grid technology and its use in e-Science (large scale distributed scientific research). Here we make the ideas more concrete by describing how one can "convert" (build from scratch) a scientific resource (program) as a web service. Modern Grids are built on top of web services with interesting refinements captured as OGSA – Open Grid Service Architecture. The approach here can easily be extended to be OGSA (with its initial OGSI standard) compliant [3].

Web-enabled applications in support of e-business and e-commerce are an everyday fact of life: customizable information portals like Yahoo, online ordering systems like Amazon, and web auctions like E-bay are familiar to everyone. The potential for web-enabling science applications has attracted a lot of attention from the scientific community as well. Numerous browser-based computing portal systems and problem solving environments have been built since the late '90s, and a comprehensive review may be found in [1]. Various commodity technologies from the world of electronic commerce, including CGI-based Perl and Python scripts, Java applets and servlets, CORBA, and most recently, Web Services, have all been brought to bear on the science portal/service problem.

This article discusses the general architecture of science portal/service systems and illustrates with a simple example how one may build a constituent service out of a particular application. To make the presentation concrete, we will develop a simple wrapper application for a code, *Disloc* [2], which is used in earthquake simulation to calculate surface displacements of observation stations for a given underground fault. Disloc is a particularly useful example for our purposes, since the code performs the calculations quickly, so we can provide this as an anonymous service, and issues such as computer accounts, allocations and scheduling are not important.

The Big Picture: Services, Portals, and Grids

Before describing the details of Web service creation, we wish to present a comprehensive view, Figure 1, of how Web Services, Portals, Grids, and hardware infrastructure are related. Going from right to left, we start with the hardware resources: computing, data storage/sources, and scientific instruments. These resources may be bound into a *computing grid* [3] through common invocation, security, and information systems. Access to particular resources is virtualized through the use of Grid/Web services. These services are in turn accessed with client applications that are built using various client-building toolkits. For computing portals, the client applications also define user interfaces using HTML for display. We may use *portlets* to collect the displays of

these various clients into aggregate portal systems, such as Jetspeed (http://jakarta.apache.org/jetspeed/site/index.html).

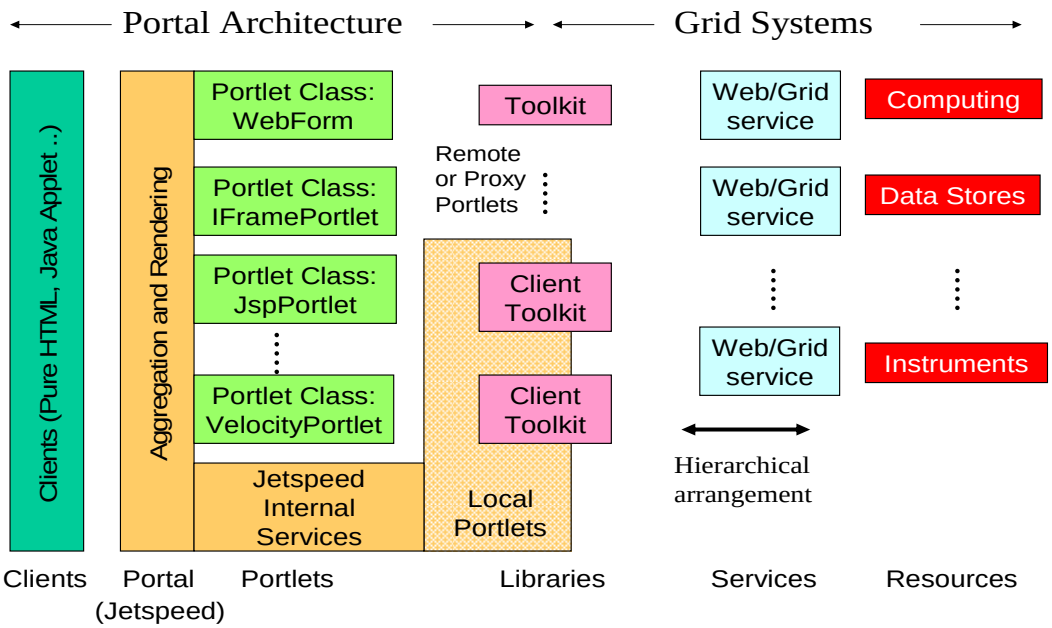

Figure 1 Aggregate portals collect interfaces to remote services.

Figure 2 shows a sample screen shot of an aggregate portal from Figure 1. The portlet on the left is a Web interface to a Disloc service (described below); the portlet on the right is an interface to a file management service for a remote host.

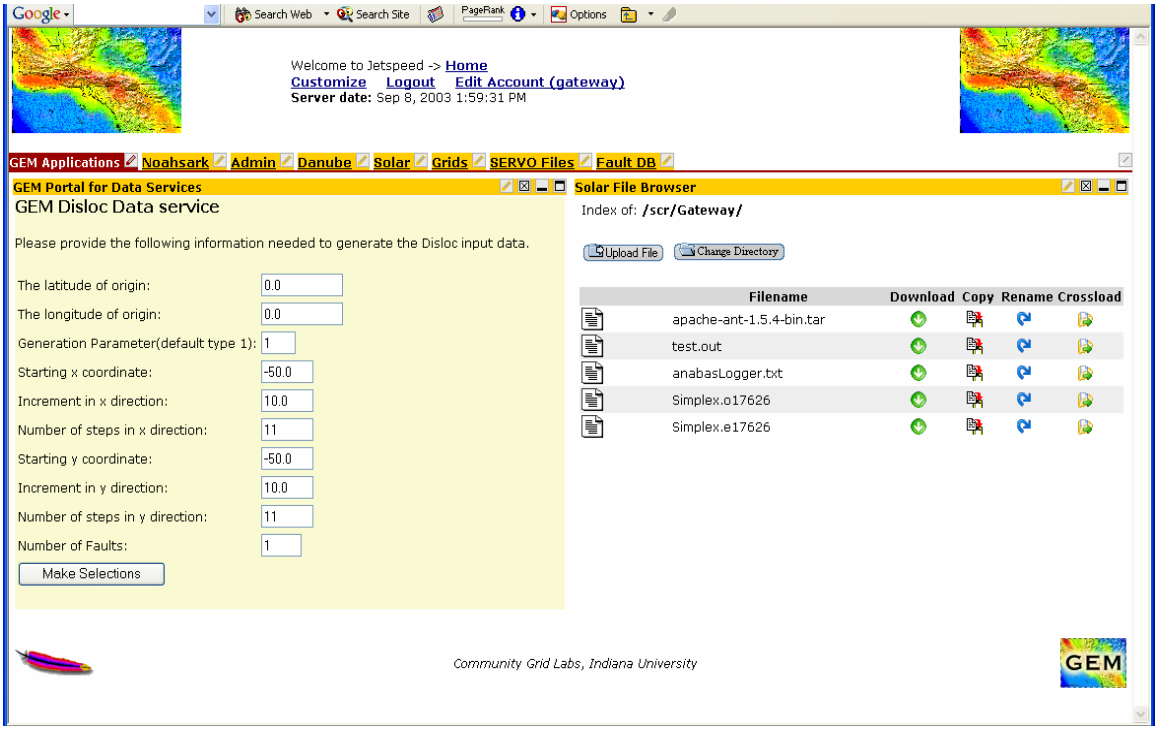

Figure 2 A screen shot of an aggregate portal.

We have discussed portals before and will return over the next year with a more detailed column on building portlet interfaces.

Web Services

What are Web Services? At its heart, the Web Service Architecture [4] is simply a system for doing distributed computing with XML-based service interface descriptions and messages. Service interfaces are described with the Web Service Description Language, WSDL. WSDL allows you to describe how to invoke a service: what are the functions, or methods, that the service provides? What arguments must I pass to the service to use the function I want to invoke? What are the argument types (integers, floats, strings, arrays) of the function and what are the return types? WSDL by itself may be thought in general terms as the XML equivalent of C header files, Java interfaces, or CORBA IDL.

The power of WSDL is that it expresses a program's interface in language-neutral XML syntax. WSDL does not directly enable remote function invocation, but does describe how to bind a particular interface to one or more remote invocation protocols. These protocols are simply ways of exchanging messages between the service provider and service invoker. Most commonly, WSDL function invocations and returns are bound to SOAP messages. These messages contain specific requests and responses: pass the subroutine *doLUDecompose* the following two-dimensional array of doubles with the following integer dimensions, and get back the LU decomposed form of the input array. When SOAP and WSDL are combined, we may build a lightly coupled system of distributed services that may be invoked and exchange information without worrying about programming language implementations or internal data structure representations.

Although Web Services may be written in any language, we stress here that Web Services are not simply CGI scripts. Web Services decouple presentation from invocation, as illustrated in Figure 1: the service component provides some specific functionality, and a client accesses it. This client is simply another program, possibly written in another programming language, which wishes to make use of a remote service. This client may also run in a Web server and generate an HTML display, but this is not required.

Science Applications as Services

While it is certainly possible to develop Web Services versions of every subroutine in *Numerical Recipes* and to rewrite all existing science applications so that they expose their functions and subroutines as services for remote components, we present a simple alternative approach that can be used to treat an entire existing application as a service. Such approaches are useful for service-enabling legacy applications and commercial codes (for which you may not have source code). This approach is also useful for wrapping applications that need to run inside of batch scheduling systems.

We will now examine (briefly) how to do this with Java. We chose Java here since there are a comprehensive set of freely available tools to build Web Service applications. The Jakarta Tomcat web server is the open-source reference implementation for the Java servlet specification. Specific web services may be built using the Java-based Apache Axis toolkit. This toolkit includes a web application that can be used to convert user-written Java applications into Web services, as well as tools that help create client boilerplate code (stubs) that simplify building clients. We note that Web services built with Java and Apache Axis can interoperate with clients written in other languages like C, and vice versa. Other (free and commercial) Web service toolkits exist for other languages, and the process of creating the Web service is roughly equivalent in the other toolkits.

Java compiles source code into byte code form, which is interpreted by a virtual machine. Calling external, non-Java programs may be done in two ways: through the Java Native Interface (to C/C++) or simply by executing an external process. Take the code *Disloc* as an example. This is written in C, but we really don't want to bother with wrapping this directly using the JNI. We instead invoke the precompiled *Disloc* executable by forking off a separate process external to the Java runtime environment. This approach sacrifices low level integration for ease of use.

The compiled copy of Disloc takes two command line arguments: the name of the input file and the name of the output file. The input file provides parameters such as the latitude and longitude of an earthquake fault and its physical dimensions and orientations, and a grid of surface observation points for the code. The output data consists of the calculated displacements of the surface grid points. Here we assume for our Disloc service that the input file has been generated and exists, or can be put, in the same file

system as the Disloc executable. Generating the input file and getting it to the right place can also be done with supporting Web Services.

Our first step is to write a Java program that can invoke the Disloc application locally. To invoke a program external to the Java Virtual Machine, one can use code such as the following:

```
public class RunExternal {
        public void runCommand(String command)  throws Exception {
                //Run the command as a process external to the
                //Java Runtime Environment.
                Runtime rt=Runtime.getRuntime();
                Process p=rt.exec(command);
        }
        …
}
```

If we compile this java program (providing a main() method, not shown), we can invoke Disloc as follows:

```
[shell> java RunExternal Disloc myinput.txt myoutput.txt
```

In practice, we would modify the above fragment to also capture standard output and standard error and put these strings in the return values for runCommand().

The above code fragment can now be converted into a deployed Web service, but first we note the following: for a real application, we certainly do not want to expose the runExec() method directly as a service for obvious security reasons, so we instead would make this a private method and surround it with publicly accessible methods such as *setDislocInput(), setDislocOutput(),* and *runDisloc()* that control and validate the input to the runExec() method.

The previous code fragment can be easily converted into a Web service with Apache Axis. First, one sets up a Tomcat web server and deploys the Axis web application. Our Java program can be deployed automatically into Axis by simply copying it into the Axis web application directory, renaming it as RunExternal.jws. This can be used for quick testing, but more formal deployment should be based around a Web Service Deployment Descriptor (WSDD), an XML file that defines the service and its allowed functions. Written descriptors can be used to deploy a Web service using Axis's AdminClient program. The interested reader should visit http://ws.apache.org/axis/.

We now have a Web Service, with methods setDislocInput(), setDislocOutput(), and runDisloc() that we expose publicly. Note that we have not written any WSDL to describe this Web Service interface. This is usually done automatically by the service container (Axis in our case). You can view the generated WSDL for your deployed service by pointing your browser to http://localhost:8080/axis/services/RunExec?wsdl. When you see it, you will be glad that you did not have to write any WSDL.

Creating Clients for Web Services

We're now ready to write clients to our Disloc Web Service. As we have emphasized previously, clients do not need to use Axis tools and do not even need to be written in Java. The general process is

1. Find/discover the service's WSDL file. You may know this already because you wrote the service, you may know it because your collaborators sent you the WSDL's URL or emailed it to you, or you may have discovered it in some online Web Service registry.
2. Write a client program that generates messages that agree with the WSDL interface. Typically these messages will be written in SOAP.
3. Send those messages to the Web Service's deployment URL. The Web Service container (Axis in our example) will inspect the SOAP message, invoke the service methods, get the results (if any), and route them back to the client.

Writing the code for clients for Web Services (step 2) can be partially automated. One approach for clients written in object oriented languages is to map the WSDL descriptions of interfaces to *stub* classes. Instances of these stub classes can be used locally by the client as if they were local objects, but in fact they simply convert arguments passed to them into SOAP calls to remote services and return locally the remote method return values. Axis, for example, provides a tool, WSDL2Java, that creates client stubs for a given WSDL file.

Conclusion: Building on the Example

In this article we have described the simplest possible service that can be used to wrap a science application (or any other code) as a Web service. We have not addressed several other issues. First, the service typically must be coupled with security systems that ensure only authenticated, authorized usage. Another important issue is *service discovery*, by which we find the URLs and descriptions for services that meet our requirements. This is one example of an information service (which may also be a Web service). We may also wish to build information services that describe, in general, how to invoke a whole range of applications. We need in this case to encode information such as how many input and output files the application takes, the location of its executable, and so forth. We may also want to encode in our information services information necessary to run the code via scheduling systems. Finally, there is the issue of coupling our service to other services into a chain. For example, the Disloc output may be coupled with a visualization service that can be used to create images that map the output vectors over a geo-referenced point.

References

[1] G. Fox, D. Gannon and M. Thomas eds. *Concurrency and Computation: Practice and Experience*, Vol. 14, No. 13-15 (2002). Special Issue on Grid Computing Environments.

[2] Dr. Andrea Donnellan of the NASA Jet Propulsion Laboratory is the author of Disloc.

[3] The Physiology of the Grid: An Open Grid Services Architecture for Distributed Systems Integration. I. Foster, C. Kesselman, J. Nick, S. Tuecke, Open Grid Service Infrastructure WG, Global Grid Forum, June 22, 2002.

[4] D. Booth, H. Haas, F. McCabe, E. Newcomer, M. Champion, C. Ferris, and D. Orchard, “Web Services Architecture.” W3C Working Draft 8 August 2003.