Editor: Geoffrey Fox, gcf@indiana.edu

Making Scientific Applications as Web Services

By Marlon Pierce and Geoffrey Fox

WEB-ENABLED E-BUSINESS AND E-COMMERCE APPLICATIONS ARE EVERYDAY FACTS OF LIFE: EVERYONE IS FAMILIAR WITH CUSTOMIZABLE INFORMATION PORTALS LIKE YAHOO, ONLINE ORDERING SYSTEMS LIKE AMAZON, AND WEB AUCTION sites like eBay. 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.[1] Various commodity technologies from the world of e-commerce, including CGI-based Perl and Python scripts, Java applets and servlets, Corba, and, most recently, Web services, have been brought to bear on the science portal–service problem.

In past columns, we discussed Grid technology in e-science (large-scale, distributed scientific research). Here, we make the ideas more concrete by describing how to convert (or build from scratch) a scientific resource (a software program) to a Web service. Modern Grids are built on top of Web services, with interesting refinements captured as an open Grid services architecture (OGSA). You easily can extend the open Grid services infrastructure (OGSI) approach we present here to be OGSA compliant.[2]

This article discusses the general architecture of science portal–service systems and illustrates with a simple example how you could 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 (authored by Andrea Donnellan of the NASA Jet Propulsion Laboratory), which is used in earthquake simulation to calculate surface displacements of observation stations for a given underground fault. Disloc's fast calculation is a particularly useful characteristic because we can provide it as an anonymous service and not have to worry about computer accounts, allocations, and scheduling.

The Big Picture: Services, Portals, and Grids

Before describing the details of Web service creation, we must look at how Web services, portals, Grids, and hardware infrastructure relate (see Figure 1). Moving from right to left, we start with the hardware resources: computing, data storage or sources, and scientific instruments. These resources might be bound into a computing Grid[2] through common invocation, security, and information systems. Access to particular resources is virtualized through Grid and Web services. In turn, client applications built from various client-building toolkits access these services. For computing portals, the client applications also define user interfaces using HTML for display. We could use portlets to collect these various clients' displays into aggregate portal systems, such as Jetspeed (http://jakarta.apache.org/jetspeed/site/index.html) and other portlet-based products available from or planned by BEA, IBM, Oracle, and others. These and other companies participated in the recent Java Community Process, JSR 168, to define the new portlet standard (see www.jcp.org/aboutJava/communityprocess/review/jsr168/).

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 later); the portlet on the right is an interface to a remote host file management service. We have discussed portals in the past and will provide a more detailed look at building portlet interfaces in a future column.

Web Services

The Web service architecture is a system for doing distributed computing with XML-based service interface descriptions and messages.[3] We use the Web Service Description Language (WSDL) to describe service interfaces. WSDL lets you describe how to invoke a service and answers these questions:

- What are the functions, or methods, that the service provides?
- What arguments must you pass to the service to use the function you want to invoke?
- What are the argument types (integers, floats, strings, arrays) of the

Copublished by the IEEE CS and the AIP 1521-9615/04/$20.00 © 2004 IEEE

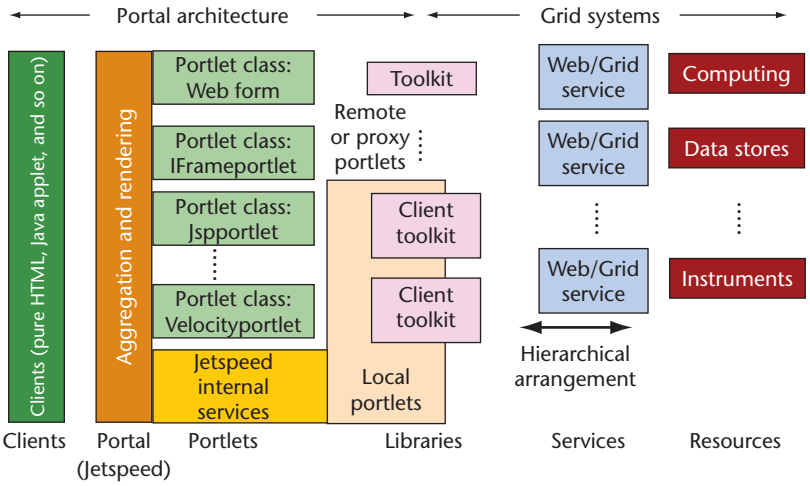

Figure 1. Aggregate portals collect interfaces to remote services. As shown, remote computing, data, and instrumental resources may be accessed by Web service-based Grids, which are, in turn, accessed through portal components via embedded client toolkits.

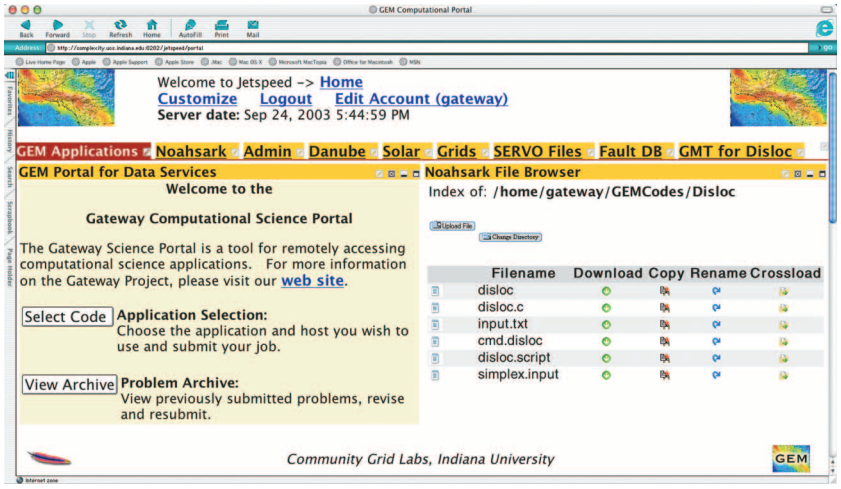

Figure 2. A screen shot of an aggregate portal. User screens are organized into tabs across the top panel. Each tab contains one or more portlets. The shown tab display contains (left) a job submission portlet and (right) a remote file management portlet.

function and what are the return types?

Think of WSDL as the XML equivalent of C header files, Java interfaces, or the Corba Interface Definition Language (IDL).

WSDL's power is that it expresses a program's interface in language-neutral XML syntax. WSDL does not directly enable remote function invocation; rather, it describes how to bind a particular interface to one or more remote invocation protocols (ways of exchanging messages between the service provider and invoker).

Usually, 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 double-precision values with the following integer dimensions, and get back the LU-decomposed form of the input array. When we combine SOAP and WSDL, we can build a lightly coupled distributed services system that can exchange information without worrying about programming language implementations or internal data structure representations.

Although we could write Web services in any language, they are not 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—that wishes to use a remote service. The client also could run in a Web server and generate an HTML display.

Science Applications as Services

Although we could develop Web services versions of every subroutine in Numerical Recipes (www.nr.com) or rewrite all existing science applications to expose their functions and subroutines as services for remote components, we'll present a simple alternative approach that you can use to treat an entire existing application as a service. This approach is useful for service-enabling legacy applications or commercial codes (for which you may not have source code) or to wrap applications to run inside batch-scheduling systems.

First, let's briefly look at how we might do this with Java. We chose Java because it has a comprehensive set of freely available tools to build Web service applications. The Jakarta Tomcat Web server (http://jakarta.apache.org/) is the open-source reference implementation for the Java servlet specification. You also can build specific Web services using the Java-based Apache Axis toolkit. This toolkit includes a Web application

```
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);
       }
       …
}
```

Figure 3. A Java program to invoke a local program. A simple Java program snippet invokes an external command. It can be embedded in a service implementation file and invoked remotely through a service invocation.

that converts user-written Java applications into Web services, as well as tools to create client boilerplate code (stubs) that simplify building clients. Java- and Apache Axis-built Web services can interoperate with clients written in other languages, such as C, and vice versa. Other (free and commercial) Web-service toolkits exist for other languages; the process of creating the Web service is roughly equivalent in these other toolkits. (Other toolkits include Java, http://xml.apache.org/axis/; XSOAP, C++ and Java toolkits for WS, www.extreme.indiana.edu/xgws/xsoap/; gSOAP, C++ SOAP toolkit, www.cs.fsu.edu/~engelen/soap.html; Python Web Services, http://pywebsvcs.sourceforge.net/; and Perl,www.soaplite.com/.)

Java compiles source code into bytecode form, which a virtual machine interprets. We could call external, non-Java programs in two ways: through the Java Native Interface (JNI)—to C/C++—or by executing an external process. For example, Disloc is written in C, but we really don't want to bother wrapping it directly using JNI. Instead, we invoke the precompiled Disloc, which is 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 commandline arguments: input and output file names. The input file provides parameters such as an earthquake fault's latitude and longitude 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. For our Disloc service, we assume that the input file exists, or can be placed, in the same file system as the Disloc executable. You also could generate the input file and get it to the right place using supporting Web services.

Our first step is to write a Java program that can invoke the Disloc application locally. Figure 3 shows what you could write to invoke a program external to the Java Virtual Machine.

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 also would modify the fragment to capture standard output and standard error strings and put them in the return values for `runCommand()`.

We now could convert the fragment into a deployed Web service, but for security reasons in a real application, we do not want to expose the `runExec()` method directly as a service. Instead, we would make this a private method and surround it with publicly accessible methods such as `setDislocInput()`, `setDislocOutput()`, and `runDisloc()` to control and validate the `runExec()` method's input.

We are now ready to convert our stand-alone code for invoking Disloc into a Web service. First, you'd set up a Tomcat Web server and deploy the Axis Web application. You could automatically deploy our Java program into Axis by copying it into the Axis Web application directory and renaming it `aRunExternal.jws`. This is good for quick testing, but for more formal deployment, it should be based around a Web service deployment descriptor (WSDD), an XML file that defines the service and its allowed functions. You could use written descriptors to deploy a Web service using Axis's AdminClient program. See http://ws.apache.org/axis/ for more information.

We now have a Web service, with methods `setDislocInput()`, `setDislocOutput()`, and `runDisloc()`, which we expose publicly. Note that we have not written any WSDL to describe this Web service interface. This usually happens automatically via the service container (Axis in our case). You can view the generated WSDL for this deployed service at http://localhost:8080/axis/services/RunExec?wsdl. When you see it, you'll be glad that you didn't have to write any WSDL.

Creating Clients for Web Services

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

1. Find or discover the service's WSDL file. You already might know this because you wrote the service, your collaborators provided you with the WSDL's URL, or be-

cause you might have discovered it in some online Web service registry.
2. Write a client program that generates messages that agree with the WSDL interface. Typically, you might write these messages in SOAP.
3. Send the messages to the Web service's deployment URL. The Web service container will inspect the SOAP message, invoke the service methods, get the results (if any), and route them back to the client.

You can partially automate writing Web services client code (step 2). One approach for clients written in object-oriented languages is to map the WSDL descriptions of interfaces to stub classes. The client can use instances of these stub classes locally 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.

In this article, we described the simplest possible service that you can use to wrap a science application (or any other code) as a Web service. We did not address several other issues: you must couple the service with security systems that ensure only authenticated, authorized users. 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 also might be a Web service). We also might wish to build information services that describe, in general, how to invoke a whole range of applications. In this case, we must encode information such as how many input and output files the application takes, the location of its executable, and so on. We also might want to encode in our information services information necessary to run the code via scheduling systems. Finally, there is the issue of linking our service to other services to create a chain. For example, we might couple the Disloc output with a visualization service that creates images that map the output vectors over a geo-referenced point.

The Web service–aggregate portal approach allows science applications to wrapped and provided as remote services with well-defined interfaces. Groups working independently can decide how they want to build their user interfaces to the services. This approach makes it simple to host computing application services at locations where experts can maintain the codes, fix problems, and so on. Other groups can use these codes as "black box" components and without having to devote computing resources.

References

1. *Concurrency and Computation: Practice and Experience*, Special Issue on Grid Computing Environments, G. Fox, D. Gannon, and M. Thomas, eds., vol. 14, nos. 13–15. 2002.
2. I. Foster et al., "The Physiology of the Grid: An Open Grid Services Architecture for Distributed Systems Integration," Open Grid Service Infrastructure WG, Global Grid Forum, 2002.
3. D. Booth et al, "Web Services Architecture," W3C Working Draft, 8 Aug. 2003.

Marlon Pierce is a senior postdoctoral research associate at the Community Grids Lab at Indiana University. He has a PhD in physics from Florida State University. Contact him at marpierc@indiana.edu.

Geoffrey Fox is director of the Community Grids Lab at Indiana University. Contact him at gcf@indiana.edu; www.communitygrids.iu.edu/IC2.html.

Submissions: Send one PDF copy of articles and/or proposals to Francis Sullivan, Editor in Chief, fran@super.org. Submissions should not exceed 6,000 words and 15 references. All submissions are subject to editing for clarity, style, and space.

Editorial: Unless otherwise stated, bylined articles and departments, as well as product and service descriptions, reflect the author's or firm's opinion. Inclusion in *CiSE* does not necessarily constitute endorsement by the IEEE, the AIP, or the IEEE Computer Society.

Circulation: *Computing in Science & Engineering* (ISSN 1521-9615) is published bimonthly by the AIP and the IEEE Computer Society. IEEE Headquarters, Three Park Ave., 17th Floor, New York, NY 10016-5997; IEEE Computer Society Publications Office, 10662 Los Vaqueros Circle, PO Box 3014, Los Alamitos, CA 90720-1314, phone +1 714 821 8380; IEEE Computer Society Headquarters, 1730 Massachusetts Ave. NW, Washington, DC 20036-1903; AIP Circulation and Fulfillment Department, 1NO1, 2 Huntington Quadrangle, Melville, NY 11747-4502. Annual subscription rates for 2004: $42 for Computer Society members (print only) and $48 for AIP society members (print plus online). For more information on other subscription prices, see www.computer.org/subscribe or http://ojps.aip.org/cise/subscrib.html. Back issues cost $20 for members, $96 for nonmembers.

Postmaster: Send undelivered copies and address changes to Circulation Dept., *Computing in Science & Engineering*, PO Box 3014, Los Alamitos, CA 90720-1314. Periodicals postage paid at New York, NY, and at additional mailing offices. Canadian GST #125634188. Canada Post Publications Mail Agreement Number 0605298. Printed in the USA.