Interacting Data Services for Distributed Earthquake Modeling

Marlon Pierce[1],Choonhan Youn[1,2], and Geoffrey Fox[1]

[1]Community Grid Labs, Indiana University
501 N. Morton Street, Suite 224
Bloomington, IN 47404-3730
{marpierc,gcf}@indiana.edu

[2]Department of Electrical Engineering and Computer Science, Syracuse University
cyoun@ecs.syr.edu

Abstract. We present XML schemas and our design for related data services for describing faults and surface displacements, which we use within earthquake modeling codes. These data services are implemented using a Web services approach and are incorporated in a portal architecture with other, general purpose services for application and file management. We make use of many Web services standards, including WSDL and SOAP, with specific implementations in Java. We illustrate how these data models and services may be used to build distributed, interacting applications through data flow.

Introduction

This paper describes our designs and initial efforts for building interacting data services for our earthquake modeling Web portal (QuakeSim). The QuakeSim portal targets several codes, including the following.: *Disloc* produces surface displacements based on multiple arbitrary dipping dislocations in an elastic half-space; *Simplex* inverts surface geodetic displacements to produce fault parameters; *GeoFEST* is a three-dimensional viscoelastic finite element model for calculating nodal displacements and tractions; *Virtual California* simulates interactions between vertical strike-slip faults; *PARK* is a boundary element program to calculate fault slip velocity history based on fault frictional properties. A complete code list and detailed descriptions may be found at [1].

The QuakeSim portal provides the unifying hosting environment for managing the various applications listed above. We briefly describe the portal here for reference, and include a more detailed description in the following section. QuakeSim is built out of portlet containers that access distributed resources via Web services, as described in [2]. The applications are wrapped as XML objects that can provide simple interactions with the hosting environments of the codes, allowing the codes to be executed, submitted to batch queuing systems, and monitored by users through the

browser. Web services are also used to manage remote files and archive user sessions. A general view of the portal architecture is shown in Figure 1.

Going beyond simple submission and management of jobs, we must also support interactions between the portal's applications. The following motivating scenario [3] illustrates the value of code integration: InSAR satellite data produces a map of surface deformations. Simplex takes this surface data and produces models with errors of the underlying fault systems that produce the deformation. By using synthetic data, one may test the potential value of additional InSAR data. In particular, one may compare the improvement on errors in the fault models when one or two additional "look" angles are added. One additional angle may be obtained from the same satellite, collecting data in both the ascending and descending portions of the satellite's path. Two additional look angles, however, require an additional satellite. Disloc may be used to generate the necessary synthetic InSAR data. Simulation results indicate in fact that an additional data stream generates a significant improvement on the estimated error parameters of the modeled fault. However, adding a third data stream produces a much less noticeable improvement over two data streams.

Scenarios such as the above require some familiarity with the simulation codes in order to manage the code linkage. When we encounter the fully interacting system, we face the additional problem of scaling. Linking any two codes may be done in a one-time fashion, but linking multiple codes is a difficult problem greatly simplified by common data formats. Also, from the portal architecture point of view, it becomes possible to develop both general purpose tools for manipulating the common data elements and also a well-defined framework for adding new applications that will share data.

QuakeSim Portal Architecture Overview

QuakeSim is based around a Web Services model, illustrated in Figure 1. The user interacts with the portal through a web browser, which accesses a central user interface server. This server maintains several client stubs that provide local interfaces to various remote services. These remote services are described in the Web Service Description Language (WSDL) [4] and are invoked via Simple Object Access Protocol (SOAP) [5] invocations over HTTP. These services are invoked on various service-providing hosts, which may in turn interact with local or remote databases (through JDBC, or Java Database Connectivity), as shown for Hosts 1 and 3; or with local queuing environments, as shown for Host 2. WSDL and SOAP are particularly useful when dealing with XML data: WSDL method (function) declarations can take both simple (string, integer, or float) and custom XML types as arguments and return types, and SOAP messages can be used as envelopes to carry arbitrary XML payloads. Readers interested in a general Web service overview are referred to [6].

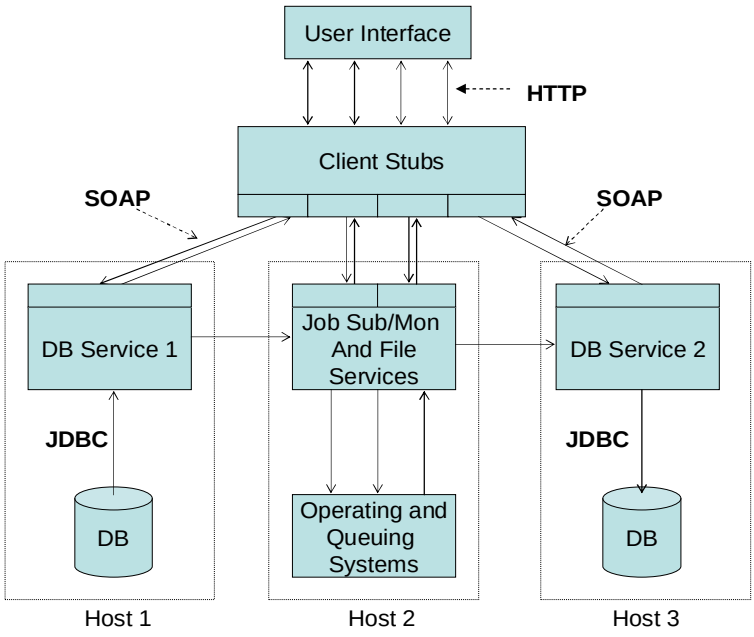

Fig. 1. QuakeSim portal architecture with Web service invocations of remote services. Arrows indicate remote invocations with indicated protocols.

The service architecture allows the portal user to browse the database on Host 1, determine the interesting data file and transfer it to Host 2. Host 2 maintains application executables, and the application may be run with the appropriate input data on a queuing system (Host 2 may be a cluster or parallel computer). Following completion of the job execution, the user may transfer the output back to another database or file system, or she may download the files to her desktop.

More detailed descriptions of the portal architecture are available from [2]. The essential pieces for file transfer and job submission and monitoring have been developed. We must now address services for managing input and output data formats for specific applications. Before proceeding, however, we wish to clarify the communication infrastructure and performance. The system we present here enables loosely couple distributions of applications and data. The primary feature of this system is the use of WSDL to describe interfaces to services. These services may be bound to more than one protocol, so if high performance file transfer (for example) is needed, we may make use of a non-SOAP implementation of a higher performance protocol. In the current system, the data sizes are sufficiently small and the inter-service communication sufficiently infrequent, that communication performance is not an issue. System time is dominated by user interactions with the system, queuing system wait times, and (in the case of Simplex) execution time. The system we have described

does not make use of inter-service communication (or intra-service communication) to during application execution. There are much more efficient communication mechanisms such as MPI for these cases, and certain of the applications we wrap (PARK, for example) make use of this.

XML Schemas for Code I/O

XML provides a useful data exchange language. It may be used to encode and provide structure to application input files and output data, and may also be transmitted easily between interacting distributed components by using SOAP. XML has several advantages for use here, but we wish to first highlight its use of namespaces. All XML schemas are named by a Uniform Resource Identifier (URI) in a structured way. We may thus be quite specific about which definitions of faults or surface deformations we are actually using within a particular XML document. We do not expect that our definitions for faults, for example, will be a final standard adopted by all, so it is useful to qualify all our definitions in this manner.

While examining the inputs and outputs for the applications to be added to QuakeSim, it became apparent that the data may be split into two portions: code-independent data definitions for fault and surface deformation data, and code-dependent formatting rules for incorporating the fault data and various code parameters, such as number of iterations and observation points. We consider as a starting case the applications Disloc and Simplex, together with fault characterization needed by all applications.

Our schema for faults is available from [8]. We highlight the major elements here. We structure our fault definitions as being composed of the following items. The *Map View* includes elements for longitude, latitude, and strike angle for the fault. The *Cartesian View* describes the location and dimensions of the fault in Cartesian space. Parameters include depth of the fault, width, length, and dip angle. *Material Properties* include various parameters needed to characterize the fault such as Lame parameters. *Slip* includes the strike slip, dip slip, and tensile components of the fault slip. Finally, *Rate* includes strike, dip, and tensile rates. Surface displacements may also be expressed in XML, as shown in [9]. Displacements may be characterized by their locations on a two-dimensional observation plane and the values of the three dimensional displacements (or rates of displacements) and errors.

Disloc and Simplex schemas may be viewed as [10] and [11], respectively. These may be compared to the actual input instructions for the codes as described in [12] and [13]. Note that we are not modifying the codes to take directly the XML input. Rather, we use XML descriptions as an independent format that may be used to generate input files for the codes in the proper legacy format. Both codes' schemas defer the definitions of Faults and Displacements to the appropriate external schema definitions, which may be included by the use of XML namespaces. The application schemas simply define the information necessary to implement the input files. The Disloc schema, for example, defines the optional format for observation points, which may be either on a regular grid or at a group of specified points.

Implementing Services for the Data Model

After defining the data models, we must next do two things: a) bind the schemas to a particular language, and b) implement services for interacting with the data through the language bindings. We first describe the general process and then look at the specific details for the Disloc, Simplex, Displacement and Fault schemas.

XML schemas map naturally to constructs in object oriented languages. For example, we may naturally map the schemas used to describe the data and code interfaces to Java data classes: each element of the schema has corresponding accessor (get and set) methods for obtaining and manipulating values. We do this (as a matter of course) with tools such as Castor [17], which automates the Java-to-XML translations.

The generated Java language bindings for the Fault, Disloc, and Simplex schemas are manipulated through service implementation files. This follows the same procedure described in [2] for Application Web Service schema. Essentially the service developer defines the programming interface she wants to expose and wraps the corresponding Java data class method invocations. This interface serves essentially as a façade to simplify interactions with the generated Castor classes. The service provider then translates the service interface into WSDL and "publishes" this interface. Publication may be done informally or through information services such as WSIL [14].

Client user interfaces for creating and manipulating the remote service may be built in the following manner: the developer downloads the WSDL interface and generates client-side stubs. The stubs are proxy objects that may be used as if they were local objects, but which invoke remote methods on the server. The client developer creates a particular interface based around the general purpose client stubs.

We now follow the above procedures for creating services for managing Disloc input and output. We describe the service implementation using Java interfaces and abstract classes, which define contracts that let an implementing class know what methods it must define for compatibility.

As described above, the Fault, Displacement, Disloc, and Simplex schemas may be converted automatically into Java classes using Castor, but we still must produce a developer-friendly façade over the data bindings. We must go a step further and define two generic interfaces that must be implemented by all applications: methods for handling Fault data and Displacement data. We also define an abstract (unimplemented) parent for code files which requires that its children implement methods for importing and exporting code data into legacy formats.

The FaultData interface fragment in Java has the following methods:

```
public interface FaultData {

          public Fault[] getAllFaults();

          public Fault getFault(String faultName);
```

```
        public void setFault(Fault sampleFault, String
faultName);

...

}
```

This interface defines general methods possessed by all implementing classes. The variable Fault and the array of Faults, Fault[], are just Java representations of the XML Fault definition. Similarly, the DisplaceData interface defines general methods for manipulating Displacements, with corresponding method names and arguments.

Finally, we require an abstract parent that, when extended by a particular application, will implement the translations between XML legacy input and output data formats of the code. For example, we may express the input file for Disloc using XML, but to generate the input file for actually running the code, we must export the XML to the legacy format. Similarly, when Disloc has finished, we must import the output data and convert its legacy format into XML, where we may for example exchange Fault or Displacement data with another application. The reverse operations must also be implemented. The GEMCode abstract parent captures these requirements:

```
public abstract class GEMCode {

        pubic abstract void exportInputData(File f,
GEMCode gc);

        public abstract GEMCode importInputData(File
filename);

        public abstract void exportOutputData(File f,
GEMCode gc);

        public abstract GEMCode importOutputData(File
filename);

}
```

This defines the method names for general important and export methods, which must be fleshed out by the implementation.

Finally, the application implementation must extend its abstract parent and implement the FaultData and DisplaceData methods. It will also need to define relevant applications methods. For example, a partial listing (in Java) for Disloc would be

```
public class DislocData extends GEMCode implements
FaultData, DisplaceData {

        public void createInputFile() { ; }
```

```
        public void setObservationStyle(String
obsvStyle) { ; }

        public void setGridObsvPoints(XSpan x, YSpan
y) { ; }

        public void setFreeObsvPoints(PointSpec[]
points) { ; }

}
```

DislocData thus is required to implement functions for manipulating Fault and Displacement data, import and export methods that translate between XML and Dislocc legacy formats, and finally Disloc-specific methods for setting observation points, etc. Note the variables XSpan, YSpan, and PointSpec (or their arrays) are just Java classes automatically generated by the data bindings from the XML schema descriptions.

The class listed above provides methods for setting the observation style and observation points for outputs, as well as material properties of the fault. The observation points for Disloc output may be either in a regular grid or on specified surface points. The last two methods may be used to access the appropriate output of the surface deformations.

The above code fragment is next converted into WSDL (which is language-independent but too verbose to list here) and may be used by client developers to create methods for remote invocation. The value of WSDL and SOAP here is again evident: the Fault class, for example, may be directly cast back into its XML form and sent in a SOAP message to the remote service.

Data Service Architecture

The final step is to define the architecture that describes how the various services must interact with each other. We first consider the case for Disloc running by itself, illustrated in Figure 2. Services are described in WSDL and SOAP is used for inter-component communication.

The browser interface gathers user code selections (“Disloc”) and a desired host for executing Disloc. The user then fills out HTML forms, providing information needed to construct the Fault and Disloc schemas. These pages are created and managed by the User Interface Server, which acts as a control point (through client stubs) for managing the remote services. These HTTP requests parameters are translated into XML by the Disloc data service, which implements the interfaces described previously. This file is then exported to the legacy format and transferred to the execution host. When the code exits, the legacy data format may be transferred back to the Data Service and imported into XML.

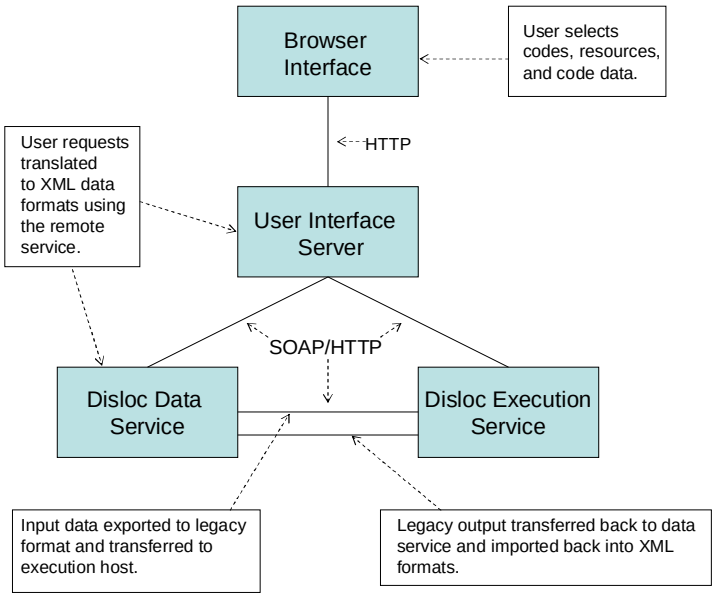

Fig. 2. Interactions of the Disloc data service. Shaded boxes indicate distributed components. Solid arrow lines indicate over-the-wire transmissions with the indicated protocols.

The value of the last step is that it now allows the output data to be shared in a common format with other applications. See for example [15] as a similar approach. Visualization and analysis services may acquire the displacement data from Disloc, as may an application such as Simplex in our motivating scenario. Such data sharing is enabled by the common XML data format, but to be used requires an additional Data Hub service, as illustrated in Figure 3.

The Data Hub service is responsible for extracting the Fault and Displacement data from the formatted XML output of applications. The Data Hub service may interoperate with other services such as the Database Service illustrated in Figure 1. Figure 3 illustrates the interaction of the Simplex and Disloc Data services with the hub.

Note that Figure 3 assumes the interactions of Figure 2 have already taken place. The User Interface server again acts as the central control point and issues commands (at either user request or through automating events) initially to the Disloc data service to transfer its output data to the hub (Step 1). The service accomplishes this in Step 2. In Step 3 the User Interface server requests that Simplex should import selected displacements from the Data Hub. The data (in XML) is then imported in Step 4. The Simplex Data Service may then (with additional Fault data) generate a Simplex legacy input file and execute the Simplex application as shown in Figure 2.

We note that other architectures are possible here. In particular, the Data Hub service is shown to act as a "push" client when accepting Disloc data (Step 2), but we may also implement it using a publish/subscribe model based around messaging systems such as the Java Messaging Service.

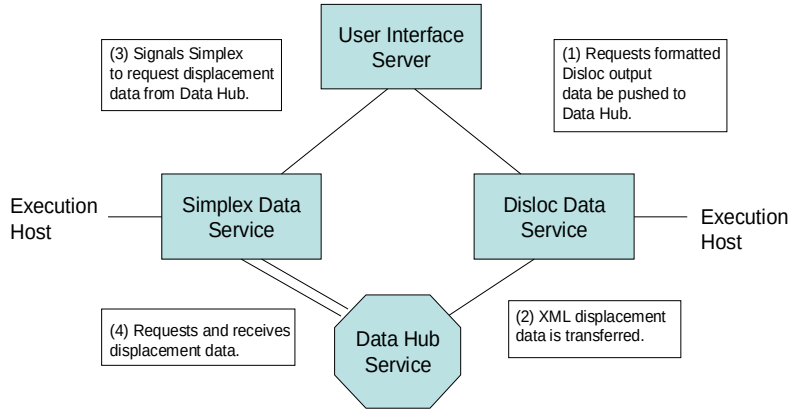

Fig. 3. Simplex and Disloc share data through the Data Hub service. Shaded objects represent distributed components. Closed arrows represent connections with execution services such as shown in Figure 2. The open arrows represent SOAP over HTTP invocations.

Summary and Conclusions

We have presented our initial architecture for implementing code specific data services. This consisted of three major steps. First, we devised XML schemas to express the application input data and code parameters. We gave examples of this process for Disloc and Simplex. Next we wrap these data models in Web services that can be plugged into our portal. These models must implement a set of specified interfaces for manipulating Faults and Displacements, as well as a parent interface that requires the service to implement import and export functions for converting between XML and legacy formats. These services may then be deployed and clients built following normal Web service development patterns. Finally, we provide a means for connecting two code-specific data services. A central data hub imports and exports XML-encoded fault and displacement data. This can be used to share, for example, synthetic Disloc displacement data with Simplex.

There are two possible revisions in our architecture. The first is in data encoding. XML is a verbose way for marking up data, and an alternate approach may be to encode only metadata in XML and use an alternative data format, such as HDF [16], for large data sets. We will need to base this on network and host performance tests

for a large number of use cases. The second possible modification is in the nature of the Data Hub. As shown in Figure 3, we have designed this to be a Web service component and assume point-to-point messaging. However, we must also explore alternative publish/subscribe mechanisms that will allow subscribed hosts to be notified when interesting data has been published. This mechanism would remove some of the low-level control capabilities from the User Interface server.

References

1. QuakeSim-System Documentation. Accessed from http://www-aig.jpl.nasa.gov/public/dus/quakesim/documentation.html
2. Pierce, M. et al.: Interoperable Web Services for Computational Portals. Proceedings of Supercomputing 2002. Available from http://sc-2002.org/paperpdfs/pap.pap284.pdf (2002)
3. Donnellan, A., et al: Final Report on the GESS Study—Inversion of Earthquake Fault Parameters Using Multiple Look Angles. NASA JPL internal report (2002).
4. Christensen, E., Curbera, F., Meredith, G., Weerawarana, S.: Web Service Description Language (WSDL) version 1.1. W3C Note 15 March 2001. Available from http://www.w3c.org/TR/wsdl.
5. Box, D., Ehnebuske, D., Kakivaya, G., Layman, A., Mendelsohn, N., Nielsen, H. F., Thatte, S., Winer, D.: Simple Object Access Protocol (SOAP) 1.1. W3C Note 08 May 2000. Available from http://www.w3.org/TR/SOAP/
6. Graham, S. et al.: Building Web Services with Java. SAMS, Indianapolis, 2002.
7. Thompson, H. S., Beech, D., Maloney, M., Mendelsohn, N. XML Schema Part 1: Structures. W3C Recommendation 2 May 2001. Available from http://www.w3.org/TR/xmlschema-1/; Biron, P. V., Malhotra, A.: XML Schema Part 2: Datatypes. W3C Recommendation 02 May 2001. Available from http://www.w3.org/TR/xmlschema-2/.
8. QuakeSim Fault Schema. Available from http://www.servogrid.org/GCWS/Schema/GEMCodes/Faults.xsd.
9. QuakeSim Surface Displacement Schema: Available from http://www.servogrid.org/GCWS/Schema/GEMCodes/Displacement.xsd.
10. QuakeSim Disloc Schema. Available from http://www.servogrid.org/GCWS/Schema/GEMCodes/Disloc.xsd.
11. QuakeSim Simplex Schema. Available from http://www.servogrid.org/GCWS/Schema/GEMCodes/Simplex.xsd.
12. Donnellan, A, et al: Elastic Dislocation Fault Parameters. Available from http://www.servogrid.org/GCWS/Schema/GEMCodes/disloc.ps.
13. Lyzenga, G. Simplex Version 4.0 Input File Format. Available from http://www.servogrid.org/GCWS/Schema/GEMCodes/simplex4.ps.
14. Ballinger, K., Brittenham, P., Malhotra, A., Nagy, W. A., Pharies, S.: Web Service Inspection Language (WS-Inspection) 1.0. IBM and Microsoft November 2001. Available from http://www-106.ibm.com/developerworks/webservices/library/ws-wsilspec.html
15. Clarke, J., Namburu, R. R.: A Distributed Computing Environment for Interdisciplinary Applications. Currency and Computation: Practice and Experience. Vol. 14, No. 13-15, p. 1161-1174 (2002).
16. NCSA HDF Home Page. Available from http://hdf.ncsa.uiuc.edu/.
17. The Castor Project. Available from http://castor.exolab.org/.