Adapting Legacy Computational Software for XMSF

Elizabeth L. White
J. Mark Pullen
Department of Computer Science
George Mason University
Fairfax, VA 22030
white@cs.gmu.edu; mpullen@gmu.edu

ABSTRACT: *The new Extensible Modeling and Simulation Framework (XMSF) shows great promise toward achieving interoperation among many previously independent software elements, using emerging Web standards and the Web services model. We have studied two different approaches to adapting existing software not previously interfaced with other models. One option is to recode the component in an object-oriented language such as Java. The second option is to wrap the legacy code with bindings for one of these languages. In both cases, the functionality can then be exported as a Web Service. While both techniques are feasible, our limited experience to date indicates that re-use of the legacy code is more effective than trying to re-engineer the code into another language. This paper will report on the techniques we have found valuable and other experience garnered in adapting existing models to XMSF linkage.*

1. Introduction

The 2002 *Extensible Modeling and Simulation Framework* (XMSF) report [1] makes a strong case that Web-based technologies have the capability to support interoperability across the spectrum of DoD models and simulations including constructive, virtual, and live modes as well as integrating legacy simulation frameworks and the increasingly important distance-learning technologies. One of the goals of the work done by this community is to define a consistent framework for integrating the new technologies to meet future challenges in a way that also harmonizes as much as possible with legacy systems.

This paper describes preliminary work and insights into integration of legacy software systems into this framework using the Web Services model. We started this work on legacy code with a number of goals in mind. First, the approach should be easy to apply. Second, platform independence would allow the technique to be widely adapted. Finally, we wanted to use inexpensive or freely available tools. These goals led us to an integration approach centered on Java, combined with a publicly available Simple Object Access Protocol (SOAP) implementation. In Section 2, we provide a brief description of Web Services and the associated technologies that we have used. In Section 3, we describe the strategies for incorporating legacy software into the Web Services model and outline the tradeoffs. Several demonstrations of the legacy based services are described briefly in Section 4.

2. Web Services

A Web Service is a self-contained, self-describing unit of modularity for publishing and delivering XML-based digital services over the Internet. Web services are the natural extension of the concept of a *resource*. They are designed to accept messages (encoded in XML) and return replies (also encoded in XML) to these messages. A Web Service's externally visible behavior is described in terms of the syntax, semantics, and sequencing of messages exchanged between the service provider and its client. Described using an XML Schema vocabulary, a Web Service interface description document specifies a contract between the service provider and its client.

Web Services can communicate directly with other Web Services in a peer-to-peer manner, or using the traditional client/server model via an HTTP server listener. Similar to components, each Web Service publishes a discoverable interface, which can be addressed or referenced via a URI (*i.e.* a URL or URN). Web Services can be invoked over a variety of transport protocols including TCP, HTTP, SMTP and message oriented middleware.

The Web Services model, described in the following section, provides a unifying framework for combining the information sharing and document distribution aspects of the Web functionality with its role as a platform for automation and integration of businesses functions. The primary supporting technologies are described in Section 2.2.

2.1 THE WEB SERVICES MODEL

The Web Services model consists of three main components: provider, consumer and registry, as shown in Figure 1. In order to utilize a Web Service, a consumer application (possibly another Web Service) would first use a registry to discover a provider of the needed service and to learn about how to invoke the service. A consumer of a Web Service will be able to bind to a service provider without necessarily knowing the interfaces or any published binding information, as long as the request for service complies with the agreed

upon attribute-based discovery protocols.

A consumer invokes the Web Service over a network (*e.g.* the Internet or an intranet) by providing the appropriate input parameters and receiving any output parameters. Web Services technologies define the messaging standards for packing and exchanging typed information between the producer and consumer of a Web Service.

Technologies such as XML[2] and SOAP [3,4] play foundational roles in defining the notion of Web Services. These technologies are still evolving; however, given the tremendous level of commercial interest and investment by the commercial vendors, standards already exist and it is expected that the technology will mature rapidly and acquire the level of functionality and stability required for its broad deployment.

2.2 XML

XML was originally heralded as the next generation of HTML, with the ability to enable the separation of content from presentation. While this is certainly a core feature, XML is now more appropriately regarded as the universal meta-language of the Web. XML is already being used for data, content, messaging, and computing to provide point-to-point integration in a platform-neutral way. XML schemas provide a standard way to define the structure, content and semantics for XML documents. This definition also serves
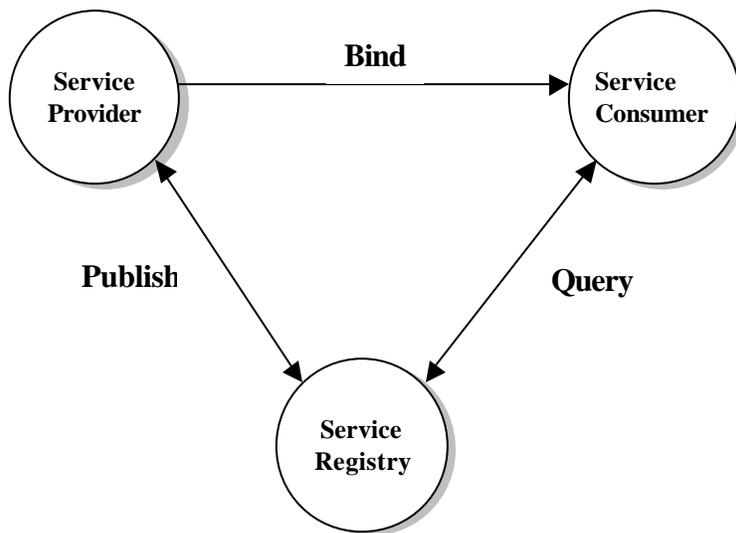

Figure 1. Web Service Components

as a specification that is used by XML parsers to validate XML documents.

The generation of lightweight, XML-based, application-level protocols now emerging ultimately will provide viable alternatives to the existing RPC-based distributed computing while relying on native Web technologies. Of these, XML, HTTP, SOAP, WSDL, and UDDI are generally regarded as mandatory pieces of the Web Services puzzle. Most of these technologies are still in the early stages of development and standardization, a process that is crucial to ensuring the interoperability of Web Services as a computing platform. XML, HTTP, and SOAP are agreed upon standards at this time, although all of these will continue to evolve.

2.3 Simple Object Access Protocol (SOAP)

The Simple Object Access Protocol (SOAP) is an XML-based, lightweight messaging protocol for exchange of typed information in decentralized, distributed environments. SOAP can enable interoperability among (existing) distributed applications running on disparate, heterogeneous platforms using a modest infrastructure. The key guiding principles in the design of SOAP are simplicity and extensibility by modularity. As such, SOAP does not define a programming model nor does it require a specific network transport. Instead, it simply consists of a modular packaging mechanism and a set of encoding rules.

SOAP provides a message format, type information and encoding mechanism that allows applications on different platforms to exchange information, irrespective of their underlying operating system, object model or programming language. The SOAP packaging mechanism allows for expressing complex communication semantics ranging from RPC-style calls to general message passing (with or without queuing). The encoding rules, on the other hand, define a data serialization format for exchanging application or platform-specific datatypes.

Encoded as a XML document, a SOAP message is a one-way transmission routed along a message path, which in addition to its final destination, can optionally identify one or more intermediaries. The root of the XML document, called the SOAP envelope, is the overall construct used for structuring the message payload and specifying its recipient(s). SOAP messages can be alternatively carried by (or late bound to) HTTP, SMTP, or TCP among other network transports. As such, SOAP messages can contain either document-oriented or procedure-oriented information. To model a specific interaction (messaging) pattern, (e.g., a request/response or a multicast pattern) SOAP messages must be grouped and routed along the appropriate specific message path where at each routing point they can be acted upon by an intermediary actor.

3. From Legacy Code to Web Services

As stated in the introduction, we started this work on generating Web Services from legacy code with a number of goals in mind:

- Easily implementable – our approach should be well described and straightforward to apply. In addition, we wanted steps that were potentially automatable[1].
- Platform independent – our approach should deal with heterogeneous platforms.
- Freely available software – tools for our approach should be low-cost.

Following these goals, the integration approach chosen centers on Java and a publicly available SOAP implementation. The resulting implementation, as shown in Figure 2, assumes a service consumer that sends and receives SOAP messages. The service provider has a Java front end that processes requests. Initially we had planned to only look at FORTRAN legacy codes; however, finding unclassified FORTRAN code that would be of interest to the simulation community provided to be more challenging than we anticipated. While we also considered how to process FORTRAN legacy code, we discovered C++ simulation code [5] of interest to the community and demonstrated our methodology with this as well.

Because most legacy code in which we were interested is not Java, the first step is to either convert the existing code to

---

[1] Although we did not address ease of automation directly, a number of the steps in the process could potentially have tool support.

Java or create a Java front end for the code. Regardless of the starting language, once we have Java-based components to work with, they are transformed into Web services as described in 3.2.
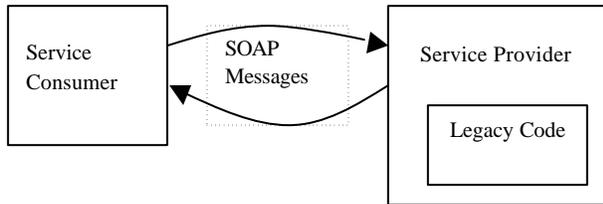

Figure 2: Java based Web Services

3.1 Legacy Code

As mentioned above, there are two feasible approaches to transforming legacy components into Java-based components. In the first approach, the legacy code can be converted to Java and then made into a Web Service. Alternatively, a Java wrapper can be created that uses the Java Native Interface (JNI) to 1) transform the input data from a Java type to one understood by the native language, 2) call the native function and 3) transform the result to a Java data type. We have successfully applied both of these approaches to FORTRAN code. For C++ code, we have only tried the second approach; however, there are tools capable of converting C++ to Java.

3.1.1 Source-to-Source Transformation of Legacy Code to Java

One way to create a Java Web service from legacy code is to reimplement the functionality in Java, either automatically via source-to-source transformation techniques or by hand. Taking this approach to producing Web Services from legacy software means that the resulting executable can be run on any architecture that supports Java, even if it does not have a compiler for the legacy language.

Our initial interest was in the transformation of legacy FORTRAN code. While there are some tools to facilitate the transformation from FORTRAN to Java, including a tool named f2j [6], we found that the process was not completely automatable for non-trivial code. For example, the tool we used did not deal with all of the data types used in FORTRAN77 code and had limitations on the control flow. For this reason, we completed the transformation by hand[2], although we did use some of the techniques described in the f2j paper, including using their mechanisms for call invocation and for common Hocks. Not surprisingly, we found manual transformations to be very time-consuming and error prone. For example, because we were dealing with FORTRAN, a language that allows implicit variable declarations, one source of problems turned out to be finding the correct location for the (approximately 350) data declarations in one of our legacy code samples.

In addition to the problems associated with manual source code transformations in general, there are several more fundamental problems. First, access to the source code is required but there are a number of reasons why this might not be possible. In addition, although the transformed code may appear to work correctly, it is difficult to be confident about the correctness without extensive testing of the Java component.

Performance is a potential weakness of this approach. The resulting Java component, unless optimized, may run slower than the original code. In the case of FORTRAN77 legacy code, this is not a surprising result considering we were comparing an interpreted program with code generated by a highly effective optimizing FORTRAN compiler.

3.1.2 Creating Java Wrappers for Legacy Code

In addition to looking at source code transformation, we looked at techniques for using the Java Native Interface (JNI) [7] to create Java wrappers that could use the native code executable directly. This approach results in a faster component, although the component itself may not be portable. An argument in favor of this approach is that the legacy code has (presumably) already been tested and this reduces the amount of testing required for the Service itself.

---

[2] Small pieces of this source to source transformation were done by writing simple specifications in Lex.

The JNI interface provides a mechanism for Java code to interact with C, C++ and (via C++) FORTRAN code. We will briefly describe the process for FORTRAN77 code that runs on Solaris; C++ code only requires part of this process. Any of several standard JNI references, including [6], provide complete details.

For the purposes of this example, assume we start with a simple FORTRAN function ADD that takes two small floating point arrays (A,B) and returns a floating point number (C).

```
REAL FUNCTION ADD *8(A,B)
REAL *8 A, B
DIMENSION A(4), B(4)
…
RETURN
END
```

Several steps are required to create a Java-based executable component that uses the legacy FORTRAN77 subroutine:

1. Write the Java wrapper object (see Appendix A), a public class that declares a native (non-Java) function with parameters types similar to that of the FORTRAN code (two floating point arrays and a floating point variable) and loads the shared library. The main function simply contains a call to the declared native function.
2. Compile the Java code, creating a class file. Next, process this class file with the javah tool to create a header file that is used in the next step.
3. Write a C++ (or C) implementation that will serve as an intermediary between the Java wrapper and the native FORTRAN code[3]. As seen in Appendix B, we call this function `add_`. In addition to declaring and invoking external function add_, this code allocates and deallocates space for the array parameters so that the legacy FORTRAN code can get to this data. It also includes two header files, one that is automatically created using the *javah* tool (as described in the second step) and a second, `jni.h`, that provides information the native code needs to call JNI functions. The parameters of add_ and the naming conventions for the C++

functions and data types are dictated by the JNI interface. .
4. Compile both the FORTRAN (using the appropriate compiler and creating an object file) and the C++ code into a dynamic shared library[4]. The name of the shared library should correspond to the name given in the Java wrapper code.

Java and C++ code for steps 1 and 3 are shown in Appendixes A and B respectively. The process is summarized in Figure 3, which focuses on the files that are created. Although there appear to be a number of steps, the most difficult part is producing the C++ and Java components. However, we believe that this code can be automatically generated given the signature of the FORTRAN code to be transformed and we are investigating this direction.

FORTRAN77 subroutines can be wrapped similarly. In this case, it will be necessary to understand the legacy code in order to determine which of the parameters have return values that need to be returned as part of the result of the Web Service. The associated C++ and Java parameters will need to be passed appropriately.

The resulting Java class component (which uses the library produced in step 4) can be run standalone; however, our interest is in providing this functionality as a Web Service. The next section describes how this component can be used as the basis for service.

3.2 Java Web Services

Transforming a Java component into a Web Service is a relatively painless process. A number of tools closely track the emerging standards for web-based interaction and they are quite straightforward to use, requiring developers to know only minimal XML and SOAP. For our implementations, we used Axis[8], which is the third generation of Apache SOAP. There are several approaches to using Axis for taking an existing Java component and transforming it into a web service. We describe the intermediate level approach briefly in this section.

---

[3] This step is required because JNI cannot link directly to FORTRAN; however, both C and C++ can.

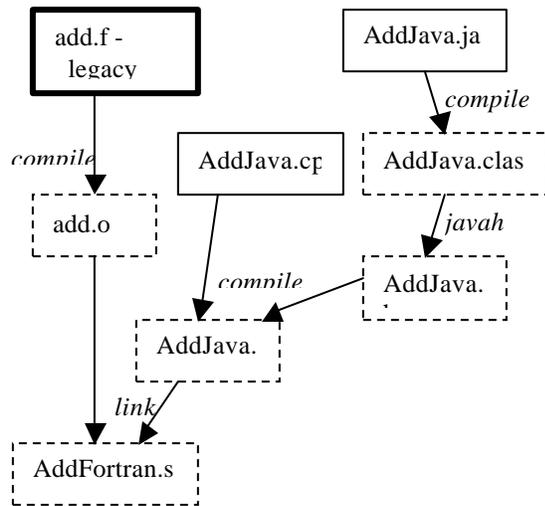[4] In Windows, one would create a dynamic link library (DLL).

Figure 3: Creating a Java Class file (AddJava.class) from FORTRAN code. Shared library AddFortran.so is used by the Java class at runtime

To deploy existing code as a Web Service, it is first necessary to write a java web service backend (JWS) file that calls the existing class. Appendix C contains a server JWS file for the example of the previous section, as well as client code. When a web request is made by a client to a Java file that is stored with a .jws extension, Axis compiles the file and invokes the Service it provides.

This method is quick, easy and does not require that the implementer know anything about SOAP and remote procedure calling. However it requires access to the source code and is difficult to debug. In addition, there is limited configuration control with this method. Axis allows more tailoring of the service using a Web Service Deployment Des criptor (WSDD). However, we have not experimented with this additional functionality.

4. Our Experiments to Date

Our focus to date has been on the feasibility of creating Web Services from legacy software and on the tradeoffs between various methodologies for this process. The work described in this paper resulted in several demonstrations. The most interesting of these is called Cannon. It was built from the C++ software package that is distributed as part of [5]. The code chosen allows the player to try to hit a target using a cannon. This simple demonstration is interactive; the player types in information about speed, height and angle in the graphical user interface. This information, including the starting location, is used to calculate a new location for the cannonball as it moves toward the target. The result is graphed and then a new calculation is done using the result from the previous invocation as the starting location. Calculations are made repeatedly until the cannonball hits the ground. At this point, the user interface reports either a hit or miss. If the user has missed, they can update the values and try again. Figure 4 shows three different tries by the player to hit the target. In each try, every point in the arc is computed by a function call.

To build this demonstration, we extracted the C++ code that computed the new location of the cannonball and converted it into a separate web service by wrapping it as a Java component. The client code was only modified by changing the original call to an Axis remote procedure call. Since this code is inside a loop, a Web Service call is made every time a new location is needed for visualizing. The server code was also wrapped in Java.

Our initial work was with FORTRAN77 legacy code. The study of source-to-source transformation mechanisms versus wrapping native code for FORTRAN was based on several pieces of FORTRAN77 legacy code, including a component of approximately 500 lines (and almost 400 variables). However, to date, we have only built demonstrations using the example code of the previous section. We have two different versions of the server code. In one version, we transformed the functionality to Java; in the other version, the FORTRAN77 code has been wrapped with C++ and Java as described in the previous section.

5. Conclusions and Future Work

This work focused specifically on integration of legacy software. Our work is still in progress however, we have had more success with wrapping legacy software than in performing transformations. We are currently building

additional demonstrations for our web services. The next steps will be to use these web services in real distributed simulations and to assess their performance and usability.

There are a number of other issues that will have to be addressed to enable secure, transparent use of Web Services in XMSF. One issue that we did not address during the course of this work was the question of what the characteristics of the software to be converted should be. In particular, module size must be considered; because the use of a web services implies overhead for the invocation, a service that could be implemented in a few lines of code would not be useful. In addition, a service needs to be relevant to a number of applications to make the time spent converting the code worthwhile.

We also have not yet addressed the issue of registries and dynamic discovery and integration of services. Clearly, this will become of increasing interest in the XMSF community as the pool of available services grows. A common XSMF registry would allow increased use of these legacy components, as well as new components.

Acknowledgements

References

[1] Brutzman, D., K. Morse, J. M. Pullen and M. Zyda, Extensible Modeling and Simulation Framework (XMSF): Challenges for Web-Based Modeling and Simulation, Naval Postgraduate School, Monterey, CA, 2002.

[2] eXtensible Markup Language (XML), http://www.w3c.org/xml

[3] Simple Object Access Protocol (SOAP), http://www.w3c.org/soap

[4] Box, D., Ehnebuske, D, Kakivaya, G., Layman, A., Mendelsohn, N., Nielsen, H.F., Thatte, S., and Winer, D.: SOAP: Simple Object Access Protocol. In MSDN Library, Microsoft, January 2001.

[5] Bourg, David M.: "Physics for Game Developers", O'Reilly & Associates Inc, 2003.

[6] Fox, G., Li, X. Qiang, Z., and Zhigang, W.: "A Prototype of FORTRAN-to-Java Converter", Concurrency - Practice and Experience, 9(11): 1047-1061, 1997.

[7] Liang, S.: "The Java™ Native Interface: Programmer's Guide and Specification, Addison Wesley Longman, Inc. 1999.

[8] Axis: http://ws.apache.org/axis

Author Biographies

Elizabeth White is an Associate Professor of Computer Science at George Mason University. She received M.S. and Ph.D. degrees in Computer Science from the University of Maryland, College Park. Her research interests include software architecture, programming languages and compilers, distributed computing, Web technologies and dynamic reconfiguration.

J. Mark Pullen is a Professor of Computer Science at George Mason University, where he heads the Networking and Simulation Laboratory in the C3I Center. He holds BSEE and MSEE degrees from West Virginia University, and the Doctor of Science in Computer Science from the George Washington University. He is a licensed Professional Engineer, Fellow of the IEEE and Fellow of the ACM. Dr. Pullen teaches courses in computer networking and has active research in networking for distributed virtual simulation and networked multimedia tools for distance education.
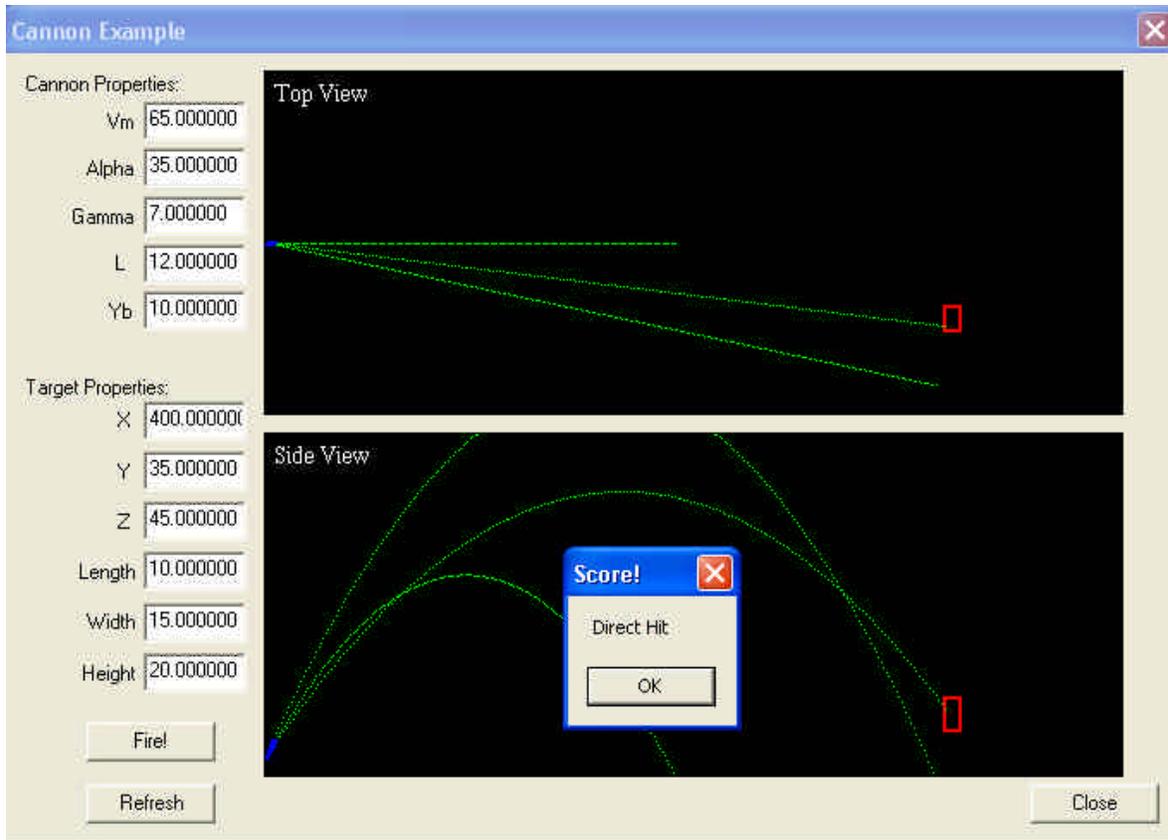
Figure 4. Cannon Demonstration

Appendix A: Java Code

```java
package test;
public class AddJava
{
  public double call_addArray(  double[] array1,
       double[] array2) {
    // create a AddJava object instance
    double sum = addArray(array1,array2);
    return sum;
  }
        // native method declaration
  public native double addArray(double[]Aarray,
    double[] Barray);
        // Load DLL (or shared library) which
       // contains implementation of native methods
       static
       {
  System.load("libaddFortran2.so");          }
}
```

Appendix B: C++ code

```cpp
#include <jni.h>          // JNI header file
#include "test_AddJava.h"    // generated from javah
  // declare the external function
extern "C" void add_(double[],double[]);
  // C++ function that is called from the Java.
JNIEXPORT jdouble JNICALL
    Java_test_AddJava_addArray( JINEnv *env,
     Jobject jobj, jdoubleArray jarr1,
     jdoubleArray jarr2)
{    // allocate space for the arrays
    jdouble *tPtr1 =
       env-> GetDoubleArrayElements(jarr1,0);
    jdouble *tPtr2 =
       env-> GetDoubleArrayElements(jarr2,0);
    // call the external function
    jdouble result = add_(tPtr1,tPtr2);
     // deallocate the space
    env->ReleaseDoubleArrayElements(jarr1,tPtr1,0);
    env->ReleaseDoubleArrayElements(jarr2,tPtr2,0);
    return result;
}
```

Appendix C: Axis (SOAP) Code

Web Server :

```java
import org.apache.axis.MessageContext;
/*** Web service to calculate the  sum of elements in 2
arrays. */
public class sumsrv
{
 /**
  * Calculates the sum.
  * @param msgContext    This is the
  *         Axis message processing
  *          Context.
  * @param array1          first array
  * @param array2          second array
  * @exception    Exception most likely
  *          a problem accessing
  *          the DB
  */
public static double   call_addArray(
    MessageContext msgContext,
    double[] array1, double[] array2)
  throws Exception
  {
    AddJava srv = new AddJava();
    return srv.call_addArray (array1,
      array2);
  }
}
```

Web Client :

```java
import org.apache.axis.client.ServiceClient;
/*
 * Sum web service client
*/
public class addClient {
  public static void main (String args[]) {
    /* Service URL */
 String url;
 double[] a1 = {1.1, 2.2, 3.3, 4.4};
 double[] a2 = {1.01, 2.02, 3.03, 4.04};
   if (args.length < 1 ) {
     System.out.println("Usage: java addClient url ")
    return;
   }
   try {
      url = args[0];
     /**
      * Invoke the sum web service
      */
    ServiceClient call = new ServiceClient(url);
    Object[] params = new Object[]{a1, a2};
    Double result = (Double)call.invoke("",
"call_addArray", params);
    System.out.println("Get sum: " + result.doubleValue());
  }catch((Exception e) {
     e.printStackTrace();
  }
 }
}
```