Web Service Interest Management (WSIM) Prototype

Mark Pullen, GMU
Presentation Overview

Case study: how to build a Web service
- WSIM architecture overview and issues
- Basic Web service implementation
- Extending the Web service
  - Streaming
  - Adding Web3D compression
  - Adding streaming output
  - Adding Overlay Multicast
WSIM Architecture as implemented
XC2I WSIM
Information Flow

Client

WSIM Client GUI

AC*

AOIM

AGIM

DIS Viewer

C2 to DIS (debug)

Client Data API

Server

Access ID Server*

Role request/Token

Access request/ACK

AOIM request/ACK

POB-tagged state change data

FOM-tagged state change data

DIS PDU

POB interface

RTI interface

DIS interface (debug)

Broker/Access Control*

Up: C2-tagged client data requests

Down: C2-tagged state change data

XML-tagged state change data

XML-tagged client data requests

XML-tagged client data requests AOIM filtered

XML-tagged state change data

XML-tagged client data requests AGIM transformed

XML-tagged client data requests

Thin Service with AC and compression

Integrated WSIM Server

LEGEND:

XML

XML/HTTP

XML/SOAP

Compressed XML

other

*GMU placeholders

GMU ODU SAIC

Multicast transport

HTTP transport

State Change data

Multicast groups

Integrated WSIM Server

Object ID server

XML/HTTP

XML/SOAP

Compressed XML

other

C2 schema; IM schema

C2 schema; AG schema

C2 schema
Scoping WSIM:
XC2I Potential Information Flow Estimate
(working estimates per Jim Blank)

• **Viewers**
  – Each potentially has 10000 objects viewable
  – 100 different simultaneous views maximum
  – Viewers may or may not overlap
  – A viewer that zooms out uses aggregation service such that there are no more updates per second from the service than when zoomed in

• **Federates**
  – 250 processors
  – 5000 objects per processor
  – Average update period 2.5 seconds

• **Worst-case aggregate flow:**
  
  400 K updates/s (~100 bytes each)
  40 MBytes/s = 320 Mb/s => not feasible on WAN
  (sensitive to the viewable objects and max views)
Ways to Reduce Network Impact of Viewer

• Limit scope in geographic and other dimensions
• Aggregate objects at server
• Don’t transmit movements too fine to be seen
• Decrease the viewer refresh rate to preclude network overload
  – Statically as startup parameter
  – or dynamically as necessary during execution
• Use streaming multicast for high-volume flows
# Web Service Overhead (~3000%)

## Pure Web service

<table>
<thead>
<tr>
<th>Component</th>
<th>Bytes</th>
</tr>
</thead>
<tbody>
<tr>
<td>Connect</td>
<td>136</td>
</tr>
<tr>
<td>HTTP Request Seg 1</td>
<td>1500</td>
</tr>
<tr>
<td>Client Ack 1</td>
<td>40</td>
</tr>
<tr>
<td>HTTP Request Seg 2</td>
<td>120</td>
</tr>
<tr>
<td>Client Ack 2</td>
<td>40</td>
</tr>
<tr>
<td>HTTP Response Seg 1</td>
<td>833</td>
</tr>
<tr>
<td>HTTP Response Seg 2</td>
<td>40</td>
</tr>
<tr>
<td>Client Ack for seg 1</td>
<td>40</td>
</tr>
<tr>
<td>Client Ack for seg 2</td>
<td>40</td>
</tr>
<tr>
<td>Response 1</td>
<td>40</td>
</tr>
<tr>
<td>Ack 1</td>
<td>48</td>
</tr>
<tr>
<td>Response 2</td>
<td>48</td>
</tr>
<tr>
<td>Ack 2</td>
<td>40</td>
</tr>
</tbody>
</table>

**Total Per Computation:** 2829 Bytes

**Grand Total**  
= 136 + \(350 \times 2829\)  
= 990286 Bytes

## Web service plus multicast

<table>
<thead>
<tr>
<th>Component</th>
<th>Bytes</th>
</tr>
</thead>
<tbody>
<tr>
<td>Connect</td>
<td>136</td>
</tr>
<tr>
<td>HTTP Request Seg 1</td>
<td>1500</td>
</tr>
<tr>
<td>Client Ack 1</td>
<td>40</td>
</tr>
<tr>
<td>HTTP Request Seg 2</td>
<td>175</td>
</tr>
<tr>
<td>Client Ack 2</td>
<td>40</td>
</tr>
<tr>
<td>HTTP Response Seg 1</td>
<td>835</td>
</tr>
<tr>
<td>HTTP Response Seg 2</td>
<td>40</td>
</tr>
<tr>
<td>Client Ack for seg 1</td>
<td>40</td>
</tr>
<tr>
<td>Client Ack for seg 2</td>
<td>40</td>
</tr>
<tr>
<td>Response 1</td>
<td>40</td>
</tr>
<tr>
<td>Ack 1</td>
<td>48</td>
</tr>
<tr>
<td>Response 2</td>
<td>48</td>
</tr>
<tr>
<td>Ack 2</td>
<td>40</td>
</tr>
</tbody>
</table>

**Total for setup:** 2886 Bytes

**Multicast Packet Size average** - 88 Bytes

**Grand Total**  
= 2886 + \(350 \times 88\)  
= 33686 Bytes
Web Services for XC2I

• Pro:
  – Easy to create
  – Easy to interface
  – Easy to compose
  – Use everywhere data volume is low

• Con:
  – Significant overhead
  – Don’t use for massive data flows
Multicast Server (XOM)

- Provides multicasting service over WAN
  - Couples to IP multicast on LAN
  - Minimizes traffic using optimal transfer tree
Overlay Multicast Tree

IP Multicast tree:
Multicast Server (XOM) Performance Requirement

• Assuming each viewer supports 10000 objects

• Based on the information flow estimates, a server might need to process 10000/2.5 = 4000 updates per second
  – Assumes all objects viewed are on same processor
  – one processor => one server => one viewer

• If AOIM groups are the same as cells, this also is the XOM data traffic per group
  – Control traffic is another matter
  – Also an XOM might support more than one group

• Current performance (Java prototype):
  – Processor-limited to 1800 messages/s in a multicast group
  – Expected to reach ~5000 messages/s in production
  – Would support one or more viewers with the same AOI
  – Parallel XOMs for more groups (but none over 5000 messages/s)\textsuperscript{11}
Building Prototype WSIM
Java Web Services
What is a WEB service?

• The term Web services means “services offered via the Web”.

• The process supporting a Web service is a method call that uses XML coding

• In a typical Web service scenario, an application sends a request to a service at a given URL using the SOAP protocol over HTTP. The web service receives the request, processes it, and returns a response back.
Web service architecture

Service Registry (Naming service)

Service Consumer (Client)

Service Provider (Server)

Discover Service (UDDI)

Public Service (WSDL)

Invoke Service (SOAP binding)
How to create a Web service?

General Steps to create a Web service:
1. Establish a Web service development environment
2. Write Server Web service method
3. Write Client to invoke service
4. Deploy service on a webserver
Web service environment

• Can use any HTTP Server
  – We chose Apache Tomcat
  – It is robust, secure and scalable and free

• Need a Simple Object Access Protocol (SOAP) implementation to encapsulate our XML
  – We use the Apache Axis open source SOAP
  – It is vendor neutral (only standard SOAP)

• Axis is written in Java
  – a C++ implementation of the client side of Axis is being developed.
About Axis

Axis isn't just a SOAP engine - it also includes:

- A simple stand-alone server
- A server which plugs into servlet engines such as Tomcat
- Extensive support for the Web Service Description Language (WSDL)
- Emitter tooling that generates Java classes from WSDL.
- Some sample programs, and
- A tool for monitoring TCP/IP packets
Step 1: Installing HTTP Server (Apache Tomcat)

• The Tomcat engine is a Java program. Before you can run Tomcat, you must have the JDK version 1.2 or later installed.

• Install Tomcat. For reference go to http://jakarta.apache.org/tomcat/index.html

• Tomcat 5.0.27 is the current release
Step 2: Installing Axis

- Axis must be installed on a HTTP server like Tomcat.
- To download Axis go to http://ws.apache.org/axis/index.html
- Copy the webapps/axis directory from the xml-axis distribution into the Web applications directory of your Application Server.
- Copy the .jar files associated with the JAXP 1.1 (Java API for XML Processing) XML compliant parser of your choice to the WEB-INF/lib directory under Axis
  - Most common choice is xerces.jar
Step 3: Starting the web service.

• Tomcat can be started by:
  – double clicking (for Windows)
  – running the start script (for Unix):
    `/bin/startup.sh` in the HTTP server
Step 4: Validate the Installation.

- At the start page of the webapp, http://127.0.0.1:8080/axis/ you should be able to see Apache-Axis start page
  - The port number may change depending upon the configuration
  - If you do not, then the webapp is not actually installed, or the appserver is not running.

Click on the link Validate, to validate the Installation. This will verify the needed and optional libraries. This presents the list of all components installed and the required components.

Do not proceed until all the required library files are present and the Validate page shows Axis is ready to run..
Step 5: Set the classpath.

- Classpath of Axis must include the following Java archive files:
  - Axis is compiled in the JAR file `axis.jar`
  - It implements the JAX-RPC API declared in the JAR files `jaxrpc.jar` and `saaj.jar`.
  - It needs various helper libraries, for logging, WSDL processing and introspection.

- The required JAR files are listed below:
  - `axis.jar`
  - `commons-discovery.jar`
  - `commons-logging.jar`
  - `jaxrpc.jar`
  - `log4j-1.2.8.jar`
  - `saaj.jar`
  - `saaj.jar`
  - `wsdl4j.jar`
Example Program 1: Hello World

• The HelloWorld client submits a SOAP request to Axis.

• The Axis server receives the request and forwards it as a Java call to the HelloWorld class.

• We write a Java class which the SOAP engine implements as a service, by invoking a method of the class
  – The operative method is callHello()

• The method returns the string back to Axis

• Axis converts this string into a SOAP response which is received by the client.
Write Web Service method on server side

// save this as HelloWorld.java in axis home directory
public class HelloWorld {
    public String callHello(String messageFromClient){
        return "Hello World " +
            messageFromClient + "!";
    }
} // end callHello method

} // end HelloWorld class
Client to Invoke Service
import statements

// save this as HelloWorldClient.java

// common classes to import for Web service client
import org.apache.axis.client.Call;
import org.apache.axis.client.Service;
import org.apache.axis.encoding.XMLType;
import javax.xml.namespace.QName;
import java.net.URL;
public class HelloWorldClient{
    public static void main(String args[]){
        try{
            String sURL =
            Object[] params = new Object[]{"My Name"};
            Service service = new Service();
            Call call = (Call) service.createCall();
            call.setTargetEndpointAddress(new java.net.URL(sURL));
            call.setOperationName(new QName(sURL,"callHello"));
            String returnValueFromWebService =
                (String) call.invoke(params);
            System.out.println("Web Service called. Returned " +
                returnValueFromWebService);
        } // end try
        catch(Exception exception){…}
    } // end main method
} // end HelloWorldClient class
Compiling and Deploying

- First compile the HelloWorld.java file to check for errors
  - Error messages from server compile phase are difficult to understand
- Axis will compile the service class again when it is invoked
  - Copy the HelloWorld.java file to the Axis root directory
  - rename the file as HelloWorld.jws (JWS – Java Web Service).
  - On first invocation, the .jws file is compiled and a class file is generated
  - The classpath must include necessary jar files.
What's happening in the client

```java
Service service = new Service();
Call call = (Call) service.createCall();
```

• *In above lines we create new Service and Call objects. These are the standard JAX-RPC objects that are used to store metadata about the service to invoke.*

```java
call.setTargetEndpointAddress(new java.net.URL(sURL));
```

• *Above line sets up the endpoint URL - this is the destination for the SOAP message.*
What's happening…

call.setOperationName(new QName(sURL,"callHello"));
• specifies the name of the method for the Web Service.

String returnValueFromWebService = (String) call.invoke(params);
• invokes the required service
• passing in an array of parameters - in this case just one String
• and also specifying the return type.
Sequence Diagram for a HelloWorld Program

HelloWorld Client

1. SOAP call

6. “Hello World” Soap Response to client

Tomcat (http server)

5. Http Response

2. Http Request

Axis Server

3. callHello()

4. “Hello World” String returned to Axis

HelloWorld class

Network messages

Internal calls within server
Example Program 2

// Server method--HelloWorld2
// takes two input parameters and returns a vector
import java.util.*;
public class HelloWorld2 {
    public HelloWorld2() {
    }
    // end of constructor

    public Vector InvokeHello(int size, String msg) {
        Vector v = new Vector();
        for (int i=0;i<size;i++) {
            v.addElement(msg);
        }
        return v;
    }
    // end HelloWorld2 function
} // end of HelloWorld2 class
import java.util.*;
// use import statements as in Example 1
public class HelloWorldClient2 {
    public static void main(String args[]) {
        try {
            String sURL =
                "http://csn.cs.gmu.edu:8080/bws/test/HelloWorld2.jws";
            Object[] params = new Object[] {new Integer(10), "Hello"};
            Service service = new Service();
            Call call = (Call) service.createCall();
            call.setTargetEndpointAddress(new java.net.URL(sURL));
            call.setOperationName(new QName(sURL, "InvokeHello"));
            Vector returnVector = (Vector) call.invoke(params);
            for (int i = 0; i < returnVector.size(); i++) {
                System.out.println((String) returnVector.elementAt(i));
            }
        } // end try
        catch (Exception exception) {…}
    } // end main method
} // end HelloWorldClient2 class
Some Important Notes

• To make values persist in the server, declare the variable as static.

• Restart the web server every time any change is made to the server class or a new service is added.

• Any supporting classes must be placed in the classpath of the axis server.
  • e.g. /axis-home/axis/WEB-INF/classes

• Print statements don’t work in the server side
  • write debug output to a file.
More Important Notes

• Utility tool TCPMonitor can be used to check the size of each transmission, the XML-SOAP message. To start this go to axis-home and run

```java
java org.apache.axis.utils.tcpmon
```

• Set the Listen port number and the targethost and listen when the web service is invoked.

• Make sure that the listen Port and Target port are different otherwise it will end up in a infinite loop.

• Example output follows
The XML-SOAP request (captured by TCPMonitor)

```xml
<?xml version="1.0" encoding="UTF-8"?>
<soapenv:Envelope
 xmlns:soapenv="http://schemas.xmlsoap.org/soap/envelope/"
 xmlns:xsd="http://www.w3.org/2001/XMLSchema"
 xmlns:xsi="http://www.w3.org/2001/XMLSchema-instance">
  <soapenv:Body>
    <ns1:callHello
     soapenv:encodingStyle="http://schemas.xmlsoap.org/soap/encoding/"
      <ns1:arg0 xsi:type="xsd:string">My Name</ns1:arg0>
    </ns1:callHello>
  </soapenv:Body>
</soapenv:Envelope>
```
The XML-SOAP response (captured by TCPMonitor)

<?xml version="1.0" encoding="UTF-8"?>
<soapenv:Envelope
xmlns:soapenv="http://schemas.xmlsoap.org/soap/envelope/
xmlns:xsd="http://www.w3.org/2001/XMLSchema"
xmlns:xsi="http://www.w3.org/2001/XMLSchema-instance">
  <soapenv:Body>
    <ns1:callHelloResponse
soapenv:encodingStyle="http://schemas.xmlsoap.org/soap/encoding/
      <ns1:callHelloReturn xsi:type="xsd:string">Hello World My Name!</ns1:callHelloReturn>
    </ns1:callHelloResponse>
  </soapenv:Body>
</soapenv:Envelope>
About WSDL

- WSDL (Web Service Description Language) provides a language-independent technique for describing a WEB Service to a client.
- If a WEB service is deployed as a .jws file, then the WSDL can be generated by appending ?wsdl to the URL of the WEB service.
- For the helloWorld WEB service, the wsdl file is found at the URL http://csn.cs.gmu.edu:8080/bws/test/HelloWorld.jws?wsdl
- A .wsdl file specifies the number of arguments, type of arguments, return type, port number, the URL for a given WEB Service.
- It generates three files <WSName>PortType, <WSName>SoapBinding, and <WSName> java file.
WSDL for HelloWorld WS
To Make a Web Service Persist and Process a Stream

Public class IMWebService{
    // the Web service method
    public void processMessages{
        receiveSocket =
            new DatagramSocket(receivePort)
        while(true){
            receiveSocket.receive(packet);
            byte[] packetContents = packet.getData();
            // continue processing packet…
        } // end while(true) – this method never returns!
    } // end processMessages method
} end IMWebService class
Layered Classes for Web Service

Public class IMWebService{
    byte[] messageBuffer = new byte[100];
    public void processMessages(){
        // create, initialize, and call the next layer down
        NewClass objNewClass = new NewClass();
        while(true){
            messageBuffer = getNewDataMessage();
            objNewClass.programLogic(messageBuffer);
        } // end while(true)
    } // end processMessages method
} // end IMWebService class

Public class NewClass{ ... }
Code Walkthrough
(time permitting)
For more Information

- Apache Tomcat
  http://jakarta.apache.org/tomcat/
- Axis
  http://ws.apache.org/axis/index.html
- XML
  http://www.w3.org/XML/
- SOAP
  http://www.w3c.org/TR/soap
- WSDL
  http://www.w3c.org/TR/wSDL
- XMSF website
  http://www.movesinstitute.org/xmsf
- Our website
  http://netlab.gmu.edu