Web Services Toolkit Interoperability

Ivan R. Judson,[1,2] Eric Olson,[1] Scott D. Price,[2] Thomas D. Uram[2]
[1]*Computation Institute,*
University of Chicago / Argonne National Laboratory
[2]*Mathematics and Computer Science Division,*
Argonne National Laboratory
{judson, eolson, price, turam}@mcs.anl.gov

Abstract

Web services, using WSDL and SOAP and following the WS-I's Basic Profile 1.0, have become the lingua franca for building service-oriented systems. Until recently, the development of tools for Web service took a significant amount of time, hindering the deployment and thus the adoption of Web services as a real technology. With the advent of the Web Services Interoperability Organization (WS-I) and the resulting Basic Profile specifications, which provide guidelines for interoperable Web services, toolkits can now adhere to a set of conventions that can enable interoperability. We examine five Web services toolkits and evaluate their interoperability on simple test cases. Our goal is twofold: to report on the effort needed to produce interoperable services using these five toolkits, and to be sufficiently rigorous in our method that others can replicate our results.

1. Introduction

Web services are the current technological solution to an old problem: remote access to data or programs. Recently, the adapter pattern approach [1] has provided a "wrapper interface" enabling previously disjoint network data and applications to interact with each other. The standards settled upon are XML Schemas [2, 3], SOAP, WSDL, and the Basic Profile 1.0 [4].

1.1 Building Web services: An initial attempt

Since their introduction as official activities in the W3C, SOAP [5] and WSDL [6] have become the standards for building Web services. However, many factors have caused Web services toolkits to remain proprietary and noninteroperable. The first two toolkits widely used, Microsoft's .Net [7] and Apache's Java-based toolkit [8], were well built, with many useful tools and utilities for Web services developers. However, they included "value-added" features and benefits that were implemented by using proprietary mechanisms, and often these extensions were not clearly indicated. Casual developers found themselves building Web services that were usable only by clients built with the same Web services toolkit. Worse, because features were added and modified with every release, Web services built with new versions of the same toolkit often did not work with new toolkit releases.

1.2 Steps toward interoperability

By late 2003 enough Web services had been deployed, regularly used, and relied upon that interoperability became a serious issue. Around the same time, IBM and Microsoft changed their strategy for their Web services toolkits. Instead of racing ahead with advanced features; they started developing their toolkits to be solid, reliable tools for developing Web services for users of any platform. The new strategy embraces interoperability but exposes the advanced features of the underlying environment in simple ways. This strategy, they believe, will draw more developers to their respective platforms.

During this transition, the Web Services Interoperability Organization (WS-I) was formed to facilitate interoperability work that would result in faster standardization through organizations such as the Internet Engineering Task Force (IETF), W3C, and OASIS. One of the first deliverables of the WS-I was an interoperability target called the Basic Profile 1.0 (BP-1.0), which, if adhered to, can provide a reasonably high degree of interoperability.

1.3 Evaluating Web services toolkits

Most current Web services toolkits claim to produce BP-1.0-compliant Web services by default. We have evaluated this assertion for the top five most commonly used toolkits: Microsoft's .NET, Apache's Axis for Java, SOAP::Lite for Perl [9], Zolera SOAP Infrastructure (ZSI) for Python [10], and gSOAP for C/C++ [11]. To test the interoperability of these toolkits, we implemented two simple Web services, each described by a BP-1.0-

compliant WSDL file. We then used each toolkit to generate server and client stubs. For each toolkit, we wrote implementations of the server and client that produced identical results. To measure interoperability, we looked at either the success or failure of the actual messaging between the clients and servers and also the messages passed between client and server on the network. Additionally, where interoperability was not found, we modified the incorrect toolkits, thereby enabling complete interoperability among all five toolkits.

The rest of this paper is organized as follows. In Section 2 we describe the interoperability tests. In Section 3 we present and analyze our results for the five toolkits. In Section 4 we discuss "best practices" for developing Web services. In Section 5 we conclude with a brief summary of how to build broadly interoperable Web services.

2. Interoperability tests

To test the interoperability of the Web services toolkit, we followed the contract-first development model [12], implementing two Web services: a SquareService with simple data types and a DateService service with complex data. The SquareService accepts a double value and returns its mathematical square, also as a double. The DateService provides two methods. The getCurrentTime method returns the current time, while the getDate method calculates a date. The getDate method accepts two input parameters: *someday*, a complex object representing an arbitrary date, and *offset*, a positive or negative integer. *Offset* is added to *someday*, and the resulting complex object is returned. The two services and their operations were designed to completely encompass the messages defined by BP-1.0.

Our test process involved the following steps:

1. Construct a BP-1.0-compliant service description using WSDL.
2. Generate client and server stubs for each toolkit.
3. Write server-side implementations for each toolkit.
4. Write client-side implementations for each toolkit.
5. Execute all five clients against all five services.
6. Where interoperability was not found,
    - examine SOAP messages,
    - modify noncompliant toolkit, and
    - repeat the interoperability test.

3. Interoperability Results

The results shown in Table 1 indicate that not all the toolkits worked "out of the box." The more mature Web service toolkits—Microsoft's .Net, Apache's Axis, and the gSOAP toolkit—produce interoperable, BP-1.0-compliant servers and clients. But the other toolkits—ZSI and SOAP::Lite—have incompatibilities that hinder interoperability. What is interesting is that both ZSI and SOAP::Lite are open source projects, while Axis is supported by Apache, and gSOAP is funded by research at a university. However, the similarities do not continue: SOAP::Lite is developed by a single primary developer with patches contributed by others, while ZSI has a group of developers working to make it better.

Table 1: Summary of initial interoperability

	.Net	*Axis*	*SOAP::Lite*	*ZSI*	*gSOAP*
.Net	✓	✓			✓
Axis	✓	✓			✓
SOAP::Lite			✓		
ZSI					
gSOAP	✓	✓			✓

Four factors caused interoperability to fail: issues with the SOAPAction header [13], SOAP message serialization and deserialization, automatic code generation, and WSDL support. These factors involve two types of failings. First, where specifications leave room for Web services toolkit developers to interpret the meaning of the specifications or leave aspects of the implementation undefined, the developers of the different toolkits may interpret or decide to implement things differently. Indeed, this was the case with the SOAPAction header in the SOAP messages and with the serialization and deserialization of the SOAP messages between the client and server. Considering the flexibility allowed by the SOAP specification, it is actually more surprising that the differences were minor enough to be corrected than that there were differences at all.

The second problem that hindered interoperability was the general maturity of the toolkits tested. That automatic code generation and WSDL support were not as advanced in the open source toolkits can easily be explained by the mismatch between the complexity of the Web services software stack and the relatively hard-to-find, high-quality, open source developer.

3.1. SOAP::Lite

The SOAP::Lite toolkit has been around for quite some time and is fairly well supported. However, some of the functionality and features of the commercially supported toolkits have yet to appear in SOAP::Lite. In particular, although SOAP::Lite does support the use of WSDL service descriptions, it was not robust enough to support our test services, so we generated all the necessary code by hand. Despite this major shortcoming, SOAP::Lite still provides a powerful toolkit that is easy to work with.

The SOAP::Lite documentation provides examples of how to call a service in several simple lines of code. Although that code works when both client and server are using SOAP::Lite, in most cases it does not work with other toolkits. With SOAP::Lite we encountered both differences in how the SOAPAction header is handled and serialization differences. These problems affected both the creation of the clients and the servers using SOAP::Lite. SOAP::Lite constructs the SOAPAction header by concatenating the URI to the service, a pound sign, and the method being called:

```
http://www.soaplite.com/Demo#hi
```

Because the SOAPAction header is loosely defined in the SOAP 1.1 protocol, there is a lot of variance among the different toolkits and how they construct and parse the value. Most of the other toolkits construct the value by concatenating the URI, a colon, and the method name:

```
http://www.soaplite.com/Demo:hi
```

In order for SOAP::Lite to be interoperable, we used *on_action* to change how SOAP::Lite created the SOAPAction value.

```
#!/usr/bin/perl
use SOAP::Lite;

$proxy =
'http://services.soaplite.com/hibye.cgi';

print SOAP::Lite
   -> uri('http://www.soaplite.com/Demo')
   -> on_action( sub {
    return 'http://www.soaplite.com/Demo:hi'} )
   -> proxy($proxy)
   -> hi()
   -> result;
```

The second problem we encountered was how SOAP::Lite serializes the data to be transmitted. The default serializer adds additional XML tags that are unnecessary and unspecified by the WSDL. For instance, in the SOAP message shown in example 2, the c-gensym1 block labels are not necessary, and in fact they make some other toolkits unable to parse this message.

Strictly speaking, some toolkits will accept messages with these additional tags; however, we attempted to obtain message-level interoperability by making the SOAP messages appear as close to identical as possible. Through careful modification, nesting the various properties of SOAP::Data objects within each other by hand, we were able to produce and consume SOAP messages with SOAP::Lite that were virtually identical to the other toolkits.

```
<getSquare
xmlns:namesp1="http://www.mcs.anl.gov/WebServi
ces/SquareService">
    <c-gensym1>
        100
    </c-gensym1>
</getSquare>
```

The modification of the SOAP::Lite serializer was done in the envelope method; when the processing of the message was completed, the resulting message data was then returned to the default envelope. This inline modification of the serializer makes it transparent to the end-user or developer that any message massaging is occurring.

```
<x
xmlns:namesp1="http://www.mcs.anl.gov/WebServi
ces/SquareService">100
</x>
```

Similar overriding of the SOAP::Lite message deserializer was necessary. In order for the default deserializer to recognize the SOAP message, extraneous blocks like those we removed when sending the message needed to be inserted into received messages. Again, we put our modification inside the SOAP::Lite toolkit, transparent to users, by intercepting, modifying, and returning a transformed message.

The two problems contributing to the incompatibility with SOAP::Lite were the SOAPAction header and the SOAP Message format. We successfully found modifications to SOAP::Lite that allowed it to be interoperable. However, the solution for dealing with the SOAPAction header left the toolkit such that developers who wanted to use it for interoperable services needed to know how to use the on_action method. We decided to make it easier for developers, so we modified the SOAP::Lite toolkits HTTP.pm file, at line 205, to make the colon-separated value the default format for the SOAPAction header. This eliminates the need for any SOAPAction header modification by service developers.

3.2. Zolera SOAP Infrastructure (ZSI)

The Zolera SOAP Infrastructure is a reasonably mature toolkit, with an active development community. We have been using the ZSI tools for quite some time in the Access Grid project, primarily because they are implemented in Python, which is what our project uses. The ZSI toolkit does have some shortcomings, however, so we have been aggressive in developing solutions and making sure they get incorporated into the project.

ZSI comes with two tools that create code skeletons for both clients and servers automatically from a WSDL service description: wsdl2py and wsdl2dispatch. While these tools are extremely helpful, they can have problems parsing some service descriptions, resulting in the two programs crashing or generating incomplete or incorrect code. The service description for SquareService was parsed without problems; however, the DateService service description, despite being valid WSDL, caused both programs to crash.

After revising the DateService service description several times, we were able to get the two programs to generate code. What we discovered was that the getDate input parameter was not being handled correctly. We wrote our service descriptions following the "document/literal" style [14] so we created a complex type, getDateRequestType, to store both of the input parameters, *someday* and *offset*. The problem was that *someday* was itself a complex type. Since the initial generated code did not work, we had to modify several of the generated files.

```
#!/usr/bin/python

from DateService_client import
DateServiceBindingSOAP

ws =
DateServiceBindingSOAP("http://localhost:1235/D
ateServer")

date = ws.getCurrentDate("thisdoesntmatter")
```

All of the modifications we made to ZSI involved an aspect of Web services toolkits that deals with the mapping of SOAP message types, like *someday* and *offset*, to types in the programming language being used by the developer. Type-mapping, as it is called, is a difficult operation, but it is made even more difficult in Python and other dynamically typed languages (e.g., perl). The two changes that corrected ZSI's code generators were to disambiguate array type objects and to simplify the handling of simple types. We found that ZSI was treating an object as an array, or an array as an object, when it wasn't. The problem with simple types was that ZSI was overzealous—creating large, complex objects for simple types such as strings. We also made some changes to simplify the use of the generated code for the developers, but these were merely aesthetic, not functional, modifications of the toolkit.

When the generated code from code generators is correct, creating Python Web services clients and servers is easy and can be accomplished in a few lines of code. We used the EchoClient example provided by ZSI as a model for our interoperability test.

4. Interoperability best practices

Developing Web services can be a daunting task for even an experienced developer. Over the past few years, however, the Web services community has evolved and *mostly* agreed on some basic rules when developing web services—rules that, if followed, will produce the most interoperable Web services possible. These *best practices* consist of three basic rules:

1. Use *contract-first* design principles.
2. Use the document/literal form of the Web Services Description Language.
3. Follow the WS-I's Basic Profile 1.0.

When developing Web services, one should start with a WSDL (Web Services Description Language) file. Although WSDL files are simply XML, the WSDL 1.1 specification defines what valid XML can make up a service description. Much of a WSDL file is boilerplate, redundant code, which makes it tedious to write by hand. The five major components of the WSDL file are namespace declarations, types, messages, ports, and bindings.

Of the major components of the service description the types usually take the most work. It is here that the data types need to be defined so that Web services servers and clients can agree on what data is being used. Since the types element consists of just an XML schema definition, however, XSD editors can be used to create the necessary element definitions. Microsoft's Visual Studio.NET has an XSD editor that allows the user to toggle between a graphical designer and a text editor that features IntelliSense. Altova's XMLSpy is another good editor for type elements and provides validation for all sorts of XML files, ensuring no errors are accidentally introduced. Using such an editor is highly recommended because it allows the developer to create and modify XSD files quickly and easily.

The open source toolkits include no WSDL generators; each service description must be done by hand in a text editor. Here, then, the commercial tools really shine, because once the schemas are defined for the service types, creating the WSDL file is as trivial as using a typical Windows wizard. Thinktecture's WsContractFirst tool for designing types, services, and methods, now known as WSCF, is an integral part of the Microsoft Visual Studio Environment. If the VS.NET Add-In is installed, right-clicking on XSD files in the solution explorer will allow the user to select "Create WSDL Interface Description." Clicking this will launch the wizard. The user will first be prompted for the service name, XML namespace, and optional documentation. The user can then specify all his services and indicate whether they are Request/Response or One-Way (One-Way is defined as the client sending a request with out waiting for a response.) The next step lets the user associate a type with the in-and-out methods of the operations just defined. The last step allows the user to specify another

XSD location, if desired. The WSDL will then be generated automatically.

What distinguishes thinktectures's tools from Microsoft's own Web services tools is the methodology. Thinktecture follows the contract-first methodology, enabling easier development of interoperable Web services. The Microsoft default toolset follows a more tradition object modeling methodology..

The WSDL file will consist of the four elements mentioned earlier: types, message, portType, and binding. However, one more element is needed to make it complete: service. If with the user develops the service in VS.NET, the service element will automatically be added to the WSDL. However, it can just as easily be added by hand. It should look something like this:

```
<service name="yourServiceNameImplementation">
 <port
name="yourServiceName"binding="tns:bindingName"
>
    <soap:address

location="http://www.yourServiceLocation.com/"
/>
  </port>
</service>
```

The other tactic to ensure that Web services are as interoperable as possible is to follow the Basic Profile 1.0. The Basic Profile 1.0 *strongly* suggests the use of the "document/literal" form of service descriptions. We take that one step further and claim that the use of "document/literal" is necessary, but not sufficient, to ensure interoperability.

We suggest that developers remain relatively conservative in their inclusion of proposed standards, toolkits, and other extraneous solutions. Most, if not all, extensions are prototypes of new functionality that is desired in future Web services; and unless the services being developed require that functionality, it is much safer to avoid prototype tools and libraries.

5. Conclusions

We have presented a simple survey of the interoperability of the five most popular Web services toolkits. We purposefully did not test any tools that don't provide both client and server support, because we have focused on the tools that developers will use to build Web services infrastructure—and building server-side services is critical to that effort.

Our findings indicate that the best way to develop broadly interoperable Web services is to follow a contract-first design, with one of the commercially supported toolkits—Apache Axis or Microsoft's .Net—at least for now. The Web services world is moving fast, producing new specifications all the time, and there are at least two challenges and pitfalls when using the commercially supported toolkits.

First, the toolkits allow users to program an application and then "publish a Web service interface" by annotating the program; but because this is not contract-first design, it often causes problems, since the vendors assume the application will work with their client application. The toolkits, based on this assumption, take shortcuts and produce noninteroperable code.

Second, many of the emerging Web services standards appear in multiple forms, with different names, before they are approved by the appropriate standards bodies. Avoiding the use of prestandard specifications will help avoid interoperability problems in a heterogeneous Web services environment.

Acknowledgments

We acknowledge the Access Grid Project and the Python Web Services project for inspiring this survey and contributing to the development of a more interoperable Python Web services toolkit. We also acknowledge Microsoft Research's ConferenceXP project which has provided excellent feedback and resources on web services and interoperability. This material is based upon work in part supported by the National Science Foundation under Grant No. ANI-0222509, and by the U.S. Department of Energy under Contract W-31-109-Eng-38.

References

[1] "Adapter pattern," http://en.wikipedia.org/wiki/Adapter_pattern, 24 Jan. 2006.

[2] XML Schema Part 1. http://www.w3.org/TR/xmlschema-1/, 2004.

[3] XML Schema Part 2. http://www.w3.org/TR/xmlschema-2/, 2004.

[4] Basic Profile 1.0 (BP-1.0). http://www.ws-i.org/Profiles/BasicProfile-1.0-2004-04-16.html, 2004.

[5] Simple Object Access Protocol (SOAP). http://www.w3.org/TR/SOAP/, 2003.

[6] Web Services Description Language (WSDL). http://www.w3.org/TR/wsdl, 2001.

[7] Microsoft Visual Studio.NET, 2006. http://msdn.microsoft.com/vstudio/

[8] Web Services – Axis. Web Services Project. 2005. http://ws.apache.org/axis/
http://www.cmswatch.com/Feature/68

http://www.onjava.com/pub/a/onjava/2002/06/05/axis.html?page=1

[9] SOAP::Lite. 2005.
http://soaplite.com/
http://groups.yahoo.com/group/soaplite/
http://www.majordojo.com/soaplite/

[10] Python Web Services: Software - Zolera SOAP Infrastructure. 2005.
http://pywebsvcs.sourceforge.net/
http://sourceforge.net/mailarchive/forum.php?forum_id=1729

[11] gSOAP: C++ Web Services and Clients. 2005.
http://www.cs.fsu.edu/~engelen/soap.html
http://groups.yahoo.com/group/gsoap/

[12] Eric Cherng, James Duff, and Dino Chiesa. "Contract First Web Services Interoperability between Microsoft .NET and IBM WebSphere", *Microsoft Visual Studio Development Center*, October 28, 2004. http://msdn.microsoft.com/vstudio/java/interop/websphereinterop/default.aspx.

[13] SoapACTION
http://www.oreillynet.com/pub/wlg/2331?wlg=yes

[14] Russell Butek, "Which Style of WSDL Should I Use?" 2005. http://www-128.ibm.com/developerworks/webservices/library/ws-whichwsdl/