Web Services Invocation Framework (WSIF)

Matthew J. Duftler, Nirmal K. Mukhi, Aleksander Slominski and Sanjiva Weerawarana
IBM T.J. Watson Research Center
{e-mail: duftler, nmukhi, aslom, sanjiva @us.ibm.com }

August 9, 2001

Abstract

This paper proposes a framework that allows the application-programmer to program against an abstract service description, in a protocol-independent manner. We call this framework the Web Services Invocation Framework (WSIF). WSIF supplies a simple API to invoke Web Services, no matter how or where the service is provided, so long as the service is described in WSDL. WSIF enables the user to move away from the usual Web Services programming model of working directly with the SOAP APIs, and towards a model where the user interacts with abstract representations of the services. This allows the user to work with the same programming model regardless of how the service is implemented and accessed. The framework also allows new bindings to be dynamically added, and current bindings to be dynamically replaced.

1 Introduction

Contrary to popular opinion, Web Services are not just SOAP [1]. While SOAP is a very important protocol, and goes a long way towards enabling interoperability, things not accessible via SOAP can still be considered Web Services. There is another, somewhat broader, view of Web Services in which anything available over the network, with some associated description of its functional aspects, is considered a Web Service. For example, an EJB invocation using RMI/IIOP can still be thought of as a Web Service invocation.

Since Web Services can be accessed in many different ways, including, but not limited to, SOAP over HTTP and RMI/IIOP, the functional descriptions of these services can be quite different. A language which provides a uniform way to describe heterogeneous services is needed. The Web Services Description Language (WSDL [2]) is such a language.

WSDL allows for the abstract definition of the functional characteristics of a Web Service, and allows for the binding of this abstract definition to a concrete access mechanism. The abstract portion of WSDL defines the interface of the Web Service, including the operations, the message parts, and their types. The concrete portion defines how the service can be accessed using some well-known communication protocol or mechanism; it includes the service endpoints and their protocol bindings. WSDL also allows and encourages the addition of new protocol bindings. We expect that there will be additional bindings defined such as EJBs, RMI/IIOP, J2EE Connectors, local Java classes, and others.

There are multiple models for distributed computing, but CORBA [3], DCOM and RMI emerged over the last few years as the main standards. The programming model is typically based on the use of some kind of Interface Definition Language (IDL) that defines only the abstract service functionality. The number of protocol bindings is typically limited, and is enforced by

a tool that takes the IDL and generates the necessary wiring such as stubs and skeletons. This hides from the programmer the complexity of marshalling parameters and executing remote procedure calls (RPC). This is possible because the protocol is known ahead of time, and will not be changed. The generated code provides behavior similar to making local method calls, and makes the distinction between local and distributed services opaque. The object-oriented layer is typically built on top of RPC (sometimes called ORPC), and allows access to, and interaction with, distributed objects.

Web Services will probably quickly evolve to provide an OO layer. However, before this can be done it is necessary to make Web Service invocations (either RPC or one-way messages) easier to use and more protocol-independent. The usual Web Services programming model assumes SOAP as the underlying protocol, and this demands that the user know the details of a particular SOAP client-side implementation. This becomes a problem when the user wants to use a different SOAP package; it will require re-integration of the SOAP code.

In addition, if the user desires to access Web Services using a mechanism other than SOAP, a more generic set of APIs is required. The problem is, how can we operate on the level of the abstract service description, while still allowing multiple protocols to be used.

This paper proposes a framework which allows the user to operate at a higher level than the usual protocol-specific programming entails. With WSIF, the user programs to the abstract service representations, instead of programming to a specific client-side implementation of a protocol such as SOAP. WSIF is a set of APIs which provides the user with a uniform means to consume Web Services, regardless of the way in which the service is implemented or provided. The only requirements are that the service be described in WSDL and that a relevant protocol binding implementation is plugged into the framework.

WSIF supports two different approaches:

- The more traditional of the two approaches is to compile the WSDL document into a Java interface, an implementation of that interface (called the stub), and the necessary Java types, which reflect the specified WSDL document. The service can then be accessed using the generated Java stub.
- The other approach is to operate directly on arbitrary WSDL documents. The key difference is that no compilation cycle is required when using this approach.

2 WSDL as a generic service description layer

What follows is a WSDL document which contains the abstract definition of a service which allows one to retrieve stock quotes. This definition is abstract because it only describes the operations, and the types of the messages the operations receive and return. The types are described in terms of XML Schema types. The operations, in this example there is only one, are aggregated to form a portType.

```
<?xml version="1.0" ?>

<definitions targetNamespace="http://www.ibm.com/namespace/wsif/samples/stockquote-interface"
             xmlns:tns="http://www.ibm.com/namespace/wsif/samples/stockquote-interface"
             xmlns:xsd="http://www.w3.org/1999/XMLSchema"
             xmlns="http://schemas.xmlsoap.org/wsdl/">

  <message name="GetQuoteInput">
    <part name="symbol" type="xsd:string"/>
```

```
  </message>

  <message name="GetQuoteOutput">
    <part name="quote" type="xsd:float"/>
  </message>

  <portType name="StockquotePT">
    <operation name="getQuote">
      <input message="tns:GetQuoteInput"/>
      <output message="tns:GetQuoteOutput"/>
    </operation>
  </portType>

</definitions>
```

The abstract definition alone is not enough to actually consume the service. What is needed is a binding to a concrete protocol, or some similar mechanism. What follows is another WSDL document which describes two bindings of the abstract definition above; the first binding is to SOAP over HTTP, and the second is to a local Java class.

```
<?xml version="1.0" ?>

<definitions targetNamespace="http://www.ibm.com/namespace/wsif/samples/stockquote"
             xmlns:tns="http://www.ibm.com/namespace/wsif/samples/stockquote"
             xmlns:tns-int="http://www.ibm.com/namespace/wsif/samples/stockquote-interface"
             xmlns:xsd="http://www.w3.org/1999/XMLSchema"
             xmlns:soap="http://schemas.xmlsoap.org/wsdl/soap/"
             xmlns:java="http://schemas.xmlsoap.org/wsdl/java/"
             xmlns="http://schemas.xmlsoap.org/wsdl/">

  <import namespace="http://www.ibm.com/namespace/wsif/samples/stockquote-interface"
          location="stockquote-interface.wsdl"/>

  <binding name="SOAPBinding" type="tns-int:StockquotePT">
    <soap:binding style="rpc" transport="http://schemas.xmlsoap.org/soap/http"/>
    <operation name="getQuote">
      <soap:operation soapAction="http://example.com/GetTradePrice"/>
      <input>
        <soap:body use="encoded"
                   namespace="urn:xmltoday-delayed-quotes"
                   encodingStyle="http://schemas.xmlsoap.org/soap/encoding/"/>
      </input>
      <output>
        <soap:body use="encoded"
                   namespace="urn:xmltoday-delayed-quotes"
                   encodingStyle="http://schemas.xmlsoap.org/soap/encoding/"/>
      </output>
    </operation>
  </binding>
```

```
  <binding name="JavaBinding" type="tns-int:StockquotePT">
    <java:binding/>
  </binding>

  <service name="StockquoteService">
    <documentation>Stock quote service</documentation>
    <port name="SOAPPort" binding="tns:SOAPBinding">
      <soap:address location="http://localhost:8080/soap/servlet/rpcrouter"/>
    </port>
    <port name="JavaPort" binding="tns:JavaBinding">
      <java:address class="services.stockquote.Stockquote"/>
    </port>
  </service>

</definitions>
```

3 Translating WSDL port into protocol-dependent API calls

Given Web Service described in WSDL document it is necessary to translate WSDL binding and port into protocol API calls. The following is an example of using the Apache SOAP client-side protocol APIs to access the StockQuote service.

```
    // Build the call
    Call call = new Call();
    call.setTargetObjectURI("urn:xmltoday-delayed-quotes");
    call.setMethodName("getQuote");
    call.setEncodingStyleURI(encodingStyleURI);

    // Prepare the input message (parameters)
    Vector params = new Vector();
    params.addElement(new Parameter("symbol", String.class, symbol, null));
    call.setParams(params);

    // Execute the WSDL operation - here encapsulated in Call object
    Response resp = call.invoke(/* router URL */ url,
                                /* actionURI */ "http://example.com/GetTradePrice");

    // Check the response
    if(resp.generatedFault())
    {
      Fault fault = resp.getFault();
      System.out.println("Ouch, the call failed: ");
      System.out.println("  Fault Code   = " + fault.getFaultCode());
      System.out.println("  Fault String = " + fault.getFaultString());
      throw new SOAPException("Execution failed " + fault);
    }
    else
    {
```

```
    Parameter result = resp.getReturnValue();
    return ((Float) result.getValue()).floatValue();
  }
```

In the above example, observe the following sequence:

- Identify the service
- Identify the operation
- Identify the parameters
- Execute the operation
- Extract the result or a fault

The above steps will be common in every SOAP implementation and in any invocation of a service that is accessible via an RPC-oriented protocol. However, the Apache SOAP API specific nature of the code makes it difficult to generalize the client code and make it more protocol-independent.

4 Translating WSDL port into protocol-independent WSIF API calls

Below is an example of an invocation of the same service, this time using the WSIF APIs. It is worth noting that the steps are very similar. However, in this example they are protocol-independent. Even though the application-level code is not specific to any protocol, the effects of the two code samples are equivalent. This is because the implementation of the SOAP protocol binding uses the Apache SOAP APIs. Of course, that implementation could easily be replaced with one that uses a different client-side SOAP package, such as Apache AXIS.

```
Definition def = WSIFUtils.readWSDL(null, wsdlLocation);

WSIFDynamicPortFactory portFactory = new WSIFDynamicPortFactory(def, null, null);

// Get default port
WSIFPort port = portFactory.getPort();

// The user can also explicitly select a port to use.
// WSIFPort port = portFactory.getPort("SOAPPort");

// Prepare the input message
WSIFMessage input = port.createInputMessage();
input.setPart("symbol", new WSIFJavaPart(String.class, symbol));

// Prepare a placeholder for the output value
WSIFMessage output = port.createOutputMessage();

// Execute the WSDL operation
port.executeRequestResponseOperation("getQuote", input, output, null);
```

```
    // Check the response - this service does not return fault messages
    WSIFPart part = output.getPart("quote");

    return ((Float)part.getJavaValue()).floatValue();
```

WSIFPort is a key abstraction in WSIF - it is the runtime representation of a WSDL Port, which can be thought of as a service endpoint. WSIFPorts allow the execution of RPC or one-way operations against service endpoint. As long as the WSIFPort interface is implemented, WSDL operations can be executed. The WSIFPort implementation can be statically generated by some kind of WSDL compiler or it may be dynamically created based on a WSDL document (see description of dynamic providers below).

```
public interface WSIFPort
{
  public boolean executeRequestResponseOperation(String op,
                                                  WSIFMessage input,
                                                  WSIFMessage output,
                                                  WSIFMessage fault)
                                                    throws WSIFException;

  public void executeInputOnlyOperation(String op,
                                        WSIFMessage input)
                                          throws WSIFException;
}
```

So far we have described stub-less invocation. Stub-less invocation allows the WSIFPort to be created directly from a WSDL document. It is straightforward, but the user needs to be familiar with the WSIF APIs to get the port, create the messages, and execute the operations. It also requires an in-memory representation of the WSDL documents. This representation is based on WSDL4J [4] , which provides reflection capabilities for WSDL documents. For a given WSDL document, WSDL4J creates an in-memory object representation of each WSDL element and allows for easy navigation and manipulation via an object-oriented API.

Using the stub-less approach requires the user to be familiar with the WSIF APIs. For this reason, the ability to generate a stub that will implement a Java interface and will translate the methods of that interface into execution of operations on WSIFPort may be desirable. Such a WSDL compiler should also generate Java representations of the XML Schema types described in WSDL types section.

WSIF includes a stub compiler (called portType compiler) that takes as input WSDL file and generates JavaBeans to represent schema types defined in WSDL, Java interface that corresponds to WSDL portType and stub that is providing uniform access to web service hiding underlying protocol bindings. However, the generated code is not completely static - it uses dynamic providers to actually execute the operations so it is possible to replace WSIF protocol binding implementations used by the stub.

5 Dynamic Providers

WSIF gives to its users a uniform API to access WSDL-described Web Services. However to allow new protocol bindings to be developed and added to WSIF later it also contains an internal API that defines required behavior of a dynamic WSIF provider. To become a dynamic provider

it is necessary to implement WSIFDynamicProvider interface. The interface is used by WSIF runtime to convert WSDL port into a WSIFPort, which is used to execute operations.

Such a design enables protocol bindings to be swapped dynamically and even allows use of hand-coded port implementations in conjunction with pre-existing port implementations seamlessly. The WSIF user does not need to be concerned about how WSIF invocations are implemented but if necessary the user can override default behaviors allowing for flexible execution patterns.

As noted in [5] the Java programming language has unique features that allows new protocol implementations to be added dynamically to a running system. However there are some security risks involved with allowing user code to provide new protocol implementations and we intend to address those in a future version.

6 Conclusion

By allowing services to be invoked in an abstract manner, independent of protocol bindings and their implementations, WSIF allows application code that uses Web Services to remain independent of the details of the use of those Web Services. This simplifies the task of deployment of such applications, and opens the door to dynamic optimizations, such as runtime protocol choice and dynamic switching between protocols.

WSIF allows invocation through traditional stubs as well as dynamic invocations. Dynamic invocation is convenient and simplifies use of a Web Service as there are no generated stubs that have to be managed in order to access the service. Use of a stub allows the application-level code to be insulated from the details of the underlying WSIF API calls and allows the user to work with WSIF stubs in the same way they work with local Java objects. In addition, stub generation allows somewhat faster access to the service and may be required for some server environments that prevent dynamic invocation for security or other reasons. This flexibility gives the developer many options when integrating Web Services into their applications.

It is also worth noting that if an application employing WSIF is running inside a managed environment, such as an application server, the container can provide customized port factories, and can also dynamically switch the WSIF port implementations.

WSIF allows invocation ports to be created at runtime. This gives a user-defined factory the ability to supply invocation ports, making it relatively straightforward to treat legacy systems as web services, and also accelerates the process of integration of Web Services technologies within an enterprise.

WSIF makes using Web Services easy because of the short path between retrieving a WSDL document and consuming a service. With the static approach, the WSDL document is compiled to produce a stub, and the user can work with the generated stub as if it were any local Java object. With the dynamic approach, the WSDL document is interpreted, input messages are created using the user- specified arguments, and the operation is invoked.

References

[1] E. Box, D. Ehnebuske, G. Kakivaya, A. Layman, N. Mendelsohn, H. F. Nielsen, S. Thatte, D. Winer, "Simple Object Access Protocol (SOAP) 1.1", May 2000. Available at http://www.w3.org/TR/SOAP .

[2] Web Services Description Language (WSDL) 1.1 W3C Note 15 March 2001. Available at http://www.w3.org/TR/wsdl.html.

[3] Object Management Group, "CORBA 2.4.2 specification", Available at http://www.omg.org .

[4] WSDL4J. Available at http://www-124.ibm.com/developerworks/projects/wsdl4j/.

[5] B.Krupczak, K. Calvert, M. Ammar "Implementing Protocols in Java: The Price of Portability"