Web Services Invocation Framework:

A Step towards Virtualizing Components

Dieter König, Matthias Kloppmann, Frank Leymann, Gerhard Pfau, Dieter Roller

IBM Deutschland Entwicklung GmbH

Schönaicher Strasse 220

71032 Böblingen

dieterkoenig@de.ibm.com

matthias-kloppmann@de.ibm.com

ley1@de.ibm.com

gpfau@de.ibm.com

rol@de.ibm.com

Abstract: Initially, Web services had to do with communication between executables via ubiquitous Internet protocols and technologies. But in the meantime, Web services are used as virtualization layer for any kind of executables, both within an enterprise as well as across enterprises. As a consequence, any kind of protocol and communication technology must be supported to allow users to stay in the Web service paradigm when communicating with program functions. We introduce the Web Services Invocation Framework (WSIF) which provides a flexible runtime environment for the invocation of Web services. With a client interface that is independent from the actual invocation protocol, it supports an extensible set of protocol bindings, and performs the Web service invocation in a fully dynamic or completely compiled style. We describe the major properties of the architecture of WSIF, explain its extensibility concepts and describe its client programming model.

1 Introduction

As defined in [Au02], a *Web service* is a software application identified by a URI, whose interfaces and bindings are capable of being defined, described, and discovered as XML artifacts. A Web service supports direct interactions with other software agents using XML based messages exchanged via Internet-based protocols.

In order to exploit this concept for other types of service invocations as well, for example, in application integration scenarios, we extend this definition of a Web service beyond Internet-based protocols. Our objective is to create an extensible, generalized invocation architecture that supports both XML-based service bindings and any other service binding, and hides the details of these bindings from the client.

In the following chapter, we provide a high-level introduction to the language artifacts used to describe Web service interfaces and invocation protocols, and their extensions. The subsequent chapters explain the Web Services Invocation Framework architecture and the programming model for both the client and "providers", which extend the framework with new bindings. Finally, we will look at the implementation of this flexible and efficient runtime for the invocation of Web services, and give examples for supported binding extensions.

2 Web Service Description Language

Web services are described by XML documents that are compliant with the standardized Web Service Description Language (WSDL) [Ch01].

From a more general service oriented architecture point of view [Bu00], abstract service interfaces described by WSDL documents may be bound to Internet-based protocols (e.g. SOAP [Bo00]) as well as to any other service invocation protocol. The same service interface may actually have multiple bindings at the same time.

For this purpose, WSDL has several well-defined extension points for the description of such bindings. WSDL extensions are used for a description of the protocol between the client and the Web service provider, a description of the exchanged data formats, and a description of quality of service attributes of the service invocation, for example, security or reliability attributes [De00] [At00].

The standard elements of a WSDL document are the document's root element with namespace declarations (definitions), the definition of the abstract interface (types, messages, port types, operations), a concrete protocol binding (bindings, operations), and a specification of a service provider address (services, ports).

WSDL extensions can be located in the document's root element in order to provide general information for the overall document like namespace definitions. The WSDL types section may be extended to support type systems other than XSD; however, this is not in the scope of this document.

For the scope of this document, the important WSDL extension points are the WSDL binding element that provides protocol specific information for all operations, and the WSDL service element for the specification of the service provider address.

2.1 Related Work

The framework described in this document can be considered as one component of a service oriented architecture (SOA) "bus", which is responsible for additional invocation-related aspects like transaction management and security (see also [Bo00]).

Furthermore, the reader should get familiar with JAX-RPC [Ch02] as a portable and interoperable API for Web services. JAX-RPC is being developed through the Java Community Process as JSR-101. Note that JAX-RPC (like other efforts mentioned in [Bo00]) mainly concentrates on different flavors of SOAP, and does not address other protocols.

3 Web Services Invocation Framework Architecture

The major objectives for the Web services invocation architecture are to have a client interface that is independent from the concrete binding, to provide an extensible framework that supports different bindings, and to enable both the interpretation and compilation of WSDL documents.

The Web Services Invocation Framework (WSIF) is a toolkit and a middleware layer that insulates users of Web services from idiosyncrasies of protocols and technologies of communicating with the program implementing a particular Web service (see figure 1).

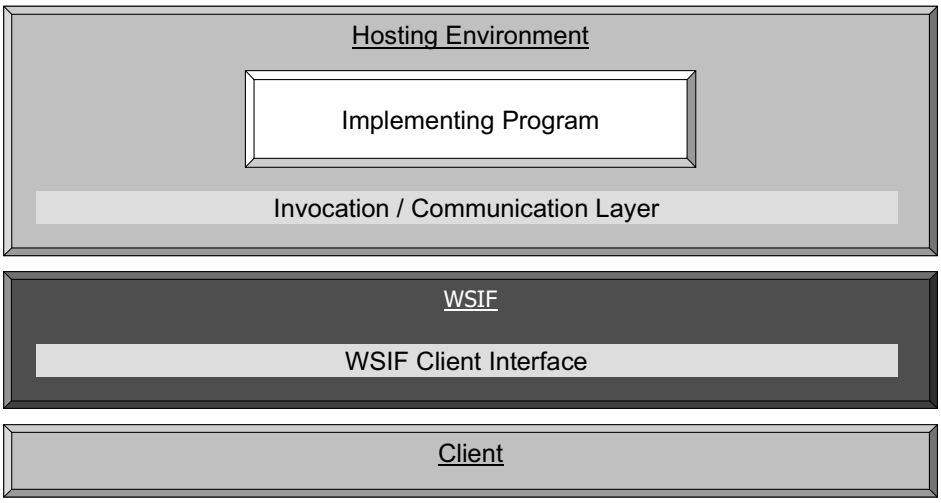

Figure 1: WSIF is the middleware layer between service binding protocols and the calling client.

The first version of WSIF was made public on IBM alphaWorks (see http://www.alphaworks.ibm.com/tech/wsif) in October 2001, and is now available as an open source project of the Apache Software Foundation (see http://xml.apache.org/axis/wsif).

In particular, WSIF provides a simple Application Programming Interface (API) that hides the service binding, a pluggable provider Service Provider Interface (SPI) for the ability to extend the runtime with additional service bindings, a dynamic lookup of invocation ports, and the ability to operate with different data type systems.

The following chapters describe how WSIF meets these architecture goals. In detail, we discuss the WSIF programming model for clients, providers, and data handling (see also figure 2). Finally, we show what implemented bindings look like, and give examples of concrete components that exploit WSIF.

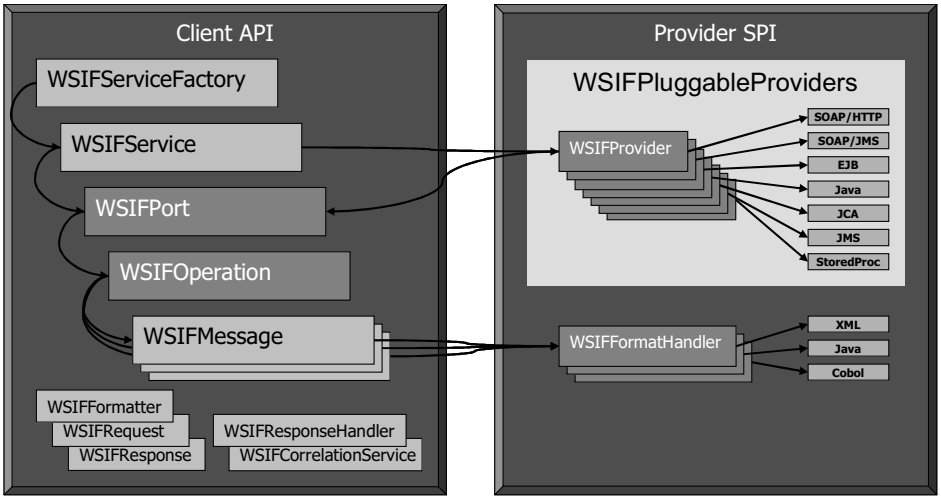

Figure 2: WSIF Architecture: the WSIF provider interface is implemented for every WSDL binding extension.

4 WSIF Programming Model

The client interaction with the WSIF runtime can be completely encapsulated in WSIF-generated stubs generated from a WSDL document, similar to the stub/skeleton generation in CORBA. Alternatively, the client may issue direct calls to the WSIF client API services, also referred to as the WSIF dynamic invocation interface (DII).

4.1 WSIF Client Interface

The WSIF dynamic invocation interface has a similar structure as the WSDL documents themselves. In other words, for many elements of WSDL documents, there are WSIF objects that represent these WSDL elements.

The WSIF client (or the generated stub) navigates through a hierarchy of WSIF factory objects, and finally calls the service invocation methods defined by the WSIFOperation interface.

The major steps performed by a WSIF client are (also reflected in the pseudo-code below) to create a service factory and use it with a WSDL document to create a service object, use the service object as a factory for a port object, use the port object as a factory for an operation object, and finally use the operation object as a factory for message objects (representing the operation's input, output, and optionally fault messages). Then, fill the input message object with the actual input data, execute the operation, and retrieve the result data from the output or fault message object, respectively.

Example: WSIF client programming model.

```
// Create WSIF service factory
WSIFServiceFactory myServiceFactory = WSIFServiceFactory.newInstance();

// Create WSIF service
WSIFService myService = myServiceFactory.getService( wsdlDefinition,
                                                     wsdlServiceNS,
                                                     wsdlService,
                                                     wsdlPortTypeNS,
                                                     wsdlPortType );
// Create WSIF port
WSIFPort myPort = myService.getPort( wsdlPort );

// Create WSIF operation
WSIFOperation myOperation = myPort.createOperation( wsdlOperation,
                                                    wsdlOperationInput,
                                                    wsdlOperationOutput );

// Create WSIF input, output, and fault messages
WSIFMessage inputMessage = myOperation.createInputMessage();
WSIFMessage outputMessage = myOperation.createOutputMessage();
WSIFMessage faultMessage = myOperation.createFaultMessage();

// Initialize WSIF input message
inputMessage.setObjectPart( inputPartName1, inputPartValue1 );
// Repeat for all input message parts ...

// Call the operation
boolean ok = myOperation.executeRequestResponseOperation( inputMessage,
                                                          outputMessage,
                                                          faultMessage );
// Process the results
if ( ok ) {
       // Process output message data ...
}
else {
       // Process fault message data ...
}
```

4.1.1 Asynchronous Invocation Protocols

For binding implementations that deal with asynchronous protocols, there exist two different styles of client interactions. Asynchronous protocols can implement an invocation of a service described as a WSDL request-response operation. Such invocations are typically divided into several phases.

First, the client sends the request message with the input data for the operation to the service provider. Then, the service provider processes the request message and returns a response message that contains either the operation's output or fault data. Finally, the client processes the response message.

The WSIFOperation interface allows the client to run all phases in one synchronous thread of execution or execute the client's part of the conversation, that is, phase 1 and 3, separately. Each phase typically also represents its own ACID transaction.

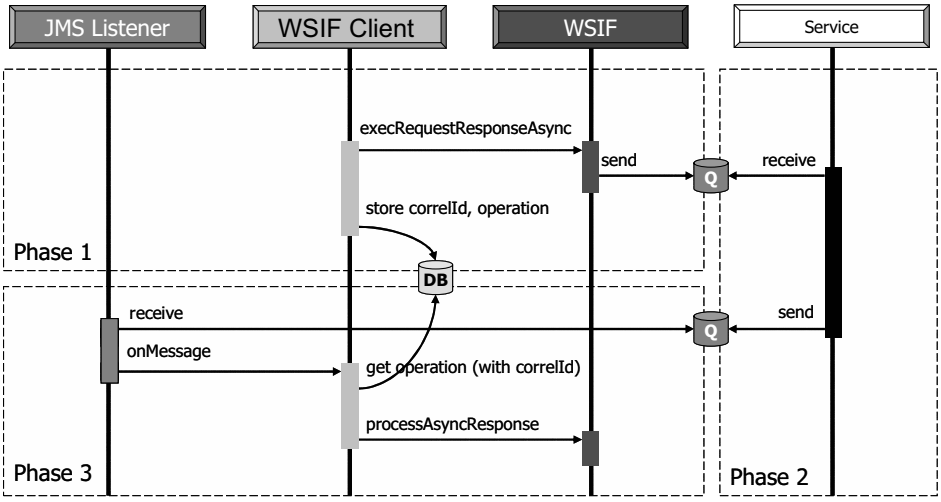

Figure 3: The asynchronous WSIF client programming model is split into separate phases.

If the synchronous WSIF interface is used, then the asynchronous behavior is completely encapsulated in a single method call. This may not be appropriate in cases where WSIF clients are not allowed to stay in a blocking wait operation until the response is received. Furthermore, the synchronous interaction is not even possible when the client is running in a transaction, because the sending of the request message must be committed in order for the message to become visible to the service provider.

If the asynchronous WSIF interface is used, then the third phase of the invocation is initiated when the response messages arrives, without the requirement for a blocking wait. If the response arrival is recognized by WSIF itself, the binding-specific wire format is processed, and the WSIF output message is returned to the client with a pre-registered response handler callback. If the binding-specific protocol listener is owned by the client, then a second WSIF method is provided for the processing of the response message (see figure 3).

4.1.2 Service Instance Reuse

The client may optionally reuse the WSIFPort object for subsequent invocations of the same or other operations that are provided by the WSDL port. In many implementations of the WSIFPort interface, the corresponding object will hold a "connection" to the service provider in order to invoke subsequent operations more efficiently by reusing existing objects, or to reuse the same service provider, in particular for stateful services.

WSIF allows serializing and deserializing the WSIFPort and its dependent objects for reuse in subsequent transactions, potentially running in a different process. Note, however, that the WSIFPort object is binding specific, and the degree of reusability is dependent on the binding properties. For examples, certain connections to service providers may only be reused within the same process.

4.2 WSIF Provider Interface

A *WSIF provider* implements a particular set of WSDL binding extensions, that is, it is in charge for hiding the idiosyncrasies of the communication with the Web service. WSIF providers have been implemented for general-purpose Web services described by the SOAP binding as well as for specialized services that are based on local method calls, on Java objects or Enterprise JavaBeans, for example.

The WSIF provider implements code that understands a set of WSDL extensions for a specific protocol binding. It is the responsibility of the WSIF provider to define a set of WSDL extensions that use a particular namespace like "soap". Additionally the WSIF provider implements a set of WSDL definition classes corresponding to the WSDL extensions. In addition to that the WSIF provider consists of provider specific implementations of the WSIFPort, WSIFOperation, and eventually WSIFMessage interfaces.

WSIF provider implementations are dynamically located using a naming convention for their respective package and class names, for example,

"org.apache.wsif.providers.soap.apachesoap"

No particular registration is required, and new providers may be dynamically added to a running system.

4.3 WSIF Data Formatting

In addition to operation bindings discussed in the previous sections, WSIF allows to handle different data format representations independent of the invocation protocol. For example, multiple protocol bindings based on Java method calls expect data to be passed as Java objects. On the other hand, protocol bindings based on message exchange may expect data to be passed in an XML document representation.

4.3.1 WSIF Format Handlers

In order to become independent from data type systems used by different invocation protocols, WSIF uses an internal data representation that does not make any assumptions about the "native" type system that is needed by the concrete WSIF provider.

The type conversion from the internal data representation to the native type system required by WSIF providers is performed by WSIF format handlers. A format handler operates on a particular data type. The description of the data type conversion is also a WSDL extension (of the binding element), and is called a *format binding*. The format binding contains "type map" elements for each data type that has to be transformed.

Attributes of the format binding and type map elements establish a naming convention that enables the WSIF framework to do a dynamic lookup of the corresponding format handler.

In the format binding, each source data type is described by an XML schema type. This is the XSD that was used in the description of the abstract operation interface.

The target data type is described by attributes of the type map element that have values which are well understood by the respective format handler.

5 WSIF Implementation

The following sections illustrate how WSDL bindings have been extended for invocation protocols that are already available with WSIF, and show how format handlers for data type transformation can be used orthogonal to the operation binding extensions.

5.1 WSIF Provider for WSDL Operation Bindings

The Apache deliverable of WSIF contains implementations of the WSIF provider interface for the following operation bindings. We show binding extensions for SOAP (using HTTP or JMS as the transport layer), Enterprise JavaBeans or native Java methods, J2EE Connector Architecture compliant connectors, and native Java Message Services (JMS).

The WSDL samples in the following sections only show the WSDL binding and service sections, and not the complete WSDL document. The description of the abstract operation interface does not have any binding-specific extension.

5.1.1 SOAP

The WSIF provider for the SOAP over HTTP binding implements the WSDL binding extensions defined by the SOAP protocol [Bo00].

Example: SOAP over HTTP.

```
<binding name="StockQuoteSoapBinding" type="tns:StockQuotePortType">
   <soap:binding style="rpc"
         transport="http://schemas.xmlsoap.org/soap/http"/>
   <operation name="GetTradePrice">
      <soap:operation soapAction="http://example.com/GetTradePrice"/>
      <input>
         <soap:body use="encoded"
               namespace=http://example.com/stockquote
               encodingStyle="http://schemas.xmlsoap.org/soap/encoding/"/>
      </input>
      <output>
         <soap:body use="encoded"
               namespace=http://example.com/stockquote
               encodingStyle="http://schemas.xmlsoap.org/soap/encoding/"/>
      </output>
   </operation>>
</binding>

<service name="StockQuoteService">
   <documentation>My first service</documentation>
   <port name="StockQuotePort" binding="tns:StockQuoteBinding">
      <soap:address location="http://example.com/stockquote"/>
   </port>
</service>
```

If the SOAP protocol is used with Java Message Services (JMS) as the transport layer, then only the *transport* attribute of the binding and the *port* element are changed.

Example: SOAP over JMS.

```
<binding name="StockQuoteSoapBinding" type="tns:StockQuotePortType">
   <soap:binding style="rpc"
         transport="http://schemas.xmlsoap.org/soap/jms"/>
   (...)
</binding>

<service name="StockQuoteService">
   <port name="StockQuotePort" binding="tns:StockQuoteBinding">
      <jms:address destinationType="queue"
           jndiConnectionFactoryName="myQCF"
           jndiDestinationName="myQ"
           initialContextFactory= "com.ibm.NamingFactory"
           jndiProviderURL= "iiop://something:900/"/>
   </port>
</service>
```

5.1.2 Enterprise JavaBeans (EJB) and Native Java

For the invocation of Enterprise JavaBeans methods, the method signature, the interface type (home/remote), and the EJB name for the directory lookup are specified.

Example: Enterprise JavaBeans.

```
<binding name="DefaultPortTypeJavaBinding" type="tns:DefaultPortType">
   <ejb:binding/>
   <operation name="readCustomer">
      <ejb:operationmapping method="getCustomer"
           partOrdering="firstName lastname customerNo"
           interface="remote" />
      <input name="RequestMessage"/>
      <output name="ResponseMessage"/>
      <fault name="FaultMessageCustomerNotFound"/>
      <fault name="FaultMessageInvalidCustomerNumber"/>
   </operation>
</binding>

<service>
   <port name="DefaultPortTypeJavaPort"
         binding="tns:DefaultPortTypeJavaBinding">
      <ejb:address class="com.ibm.example.EjbCustomerManager"
           jndiName="myapp/EjbCustomerManager" />
   </port>
</service>
```

For native Java method calls, the class definition is sufficient, that is, no lookup is required.

5.1.3 J2EE Connector Architecture (JCA)

The example below shows a WSDL binding section for a CICS J2EE Connector call. The input and output sections also contain additional specifications related to the data format passed to the CICS program (the COMMAREA data structure). This is discussed in the section about format bindings below.

Example: J2EE Connector Architecture.

```
<binding name="MyConnectorBinding"
         type="tns:MyConnectorPortType">
   <cics:binding/>
   <operation name="getCustomerInfo">
      <cics:operation functionName="CUSTINFO"/>
      <input name="Request"> ... </input>
      <output name="Response"> ... </output>
    </operation>
</binding>

<service name="MyConnectorService">
   <port name="MyConnectorPort" binding="tns:MyConnectorBinding">
      <cics:address connectionURL="xyz.ibm.com" serverName="cics21"/>
   </port>
</service>
```

5.1.4 Native Java Message Services (JMS)

The native JMS binding operates directly on the transport layer. The WSDL extensions describe the “wire format” of the JMS message. In the example, the message contains a serialized Java object.

Example: native Java Message Services.

```
<binding name="JmsBinding" type="JmsPortType">
   <jms:binding type="ObjectMessage" />
   <operation name="JmsOperation">
      <input name="RequestMessage"/>
      <output name="ResponseMessage"/>
   </operation>
</binding>

<service name="JmsService">
   <port name="JmsPort" binding="JmsBinding">
      <jms:address destinationStyle="queue"
                   jndiConnectionFactoryName="myQCF"
                   jndiDestinationName="myDestination"/>
   </port>
</service>
```

5.2 WSIF Format Handler for WSDL Format Bindings

The format binding is combined with other operation bindings shown in the previous examples, in particular those that deal with EJB, Java, and J2EE connector interfaces. It parameterizes the WSIF format handler code that is used for data type conversions.

In the following example, a format binding is combined with the native JMS binding from the previous section. In this case, the format handler presents the element described by the XSD type "**`tns:RequestObjectType`**" to the JMS provider as complex Java object of type "**`com.ibm.process.RequestObject`**". The provider will then serialize the Java object and insert it into the JMS ObjectMessage.

Example: Format Binding, embedded into the JMS binding.

```
<binding name="JmsBinding" type="JmsPortType">
   <jms:binding type="ObjectMessage" />

   <format:typemapping encoding="Java">
      <format:typemap name="tns:RequestObjectType"
                      formatType="com.ibm.process.RequestObject" />
      <format:typemap name="tns:ResponseObjectType"
                      formatType="com.ibm.process.ResponseObject" />
   </format:typemapping>

   <operation name="JmsOperation">
      <input name="RequestMessage"/>
      <output name="ResponseMessage"/>
   </operation>
</binding>

<service name="JmsService">
   <port name="JmsPort" binding="JmsBinding">
      <jms:address destinationStyle="queue"
                   jndiConnectionFactoryName="myQCF"
                   jndiDestinationName="myDestination"/>
   </port>
</service>
```

6 Summary

In this document, we introduced the Web Services Invocation Framework (WSIF), which provides a runtime environment for the invocation of Web services. We discussed the client programming model that is independent from service bindings, and the WSIF provider programming model that allows extending the framework for arbitrary bindings described with WSDL documents. Finally, we showed how WSIF is implemented and provided examples for WSDL bindings with corresponding WSIF providers in the current WSIF implementation.

6.1 Conclusion and Assessment

WSIF is a first step towards a comprehensive foundation for a service-oriented architecture, also referred to as the "SOA Bus". It has been delivered as part of the IBM WebSphere Studio Application Developer Integration Edition (WSAD-IE) Version 4.1, and is proven in practice. WSIF is used both for singular Web service invocation and in service "microflows", which are created with the WSAD-IE service flow editor. Future work should address additional bindings, performance improvements, and a close alignment with JAX-RPC.

7 References

[At00] R. Atkinson; G. Della-Libera; S. Hada; M. Hondo; P. Hallam-Baker; J. Klein; B. LaMacchia; P. Leach; J. Manferdelli; H. Maruyama; A. Nadalin; N. Nagaratnam; H. Prafullchandra; J. Shewchuk; D. Simon: Web Services Security (WS-Security) Version 1.0, IBM Corporation, Microsoft Corporation, Verisign Inc., 2000, see http://www-106.ibm.com/developerworks/library/ws-secure.

[Au02] D. Austin; A. Barbir; C. Ferris; S. Garg: Web Services Architecture Requirements, 2002, see http://www.w3.org/TR/wsa-reqs.

[Bo00] D. Box; D. Ehnebuske; G. Kakivaya; A. Layman; N. Mendelsohn; H. Nielsen; S. Thatte; D. Winer: Simple Object Access Protocol (SOAP) 1.1, W3C, 2000, see http://www.w3.org/TR/SOAP/.

[Bu00] S. Burbeck: The Tao of e-Business Services, IBM Corporation, 2000, see http://www-4.ibm.com/software/developer/library/ws-tao/index.html.

[Ch01] E. Christensen; F. Curbera; G. Meredith; S. Weerawarana: Web Service Description Language (WSDL) 1.1, W3C, 2001, see http://www.w3.org/TR/wsdl.

[Ch02] R. Chinnici et. al.: Java API for XML-Based RPC (JAX-RPC), available at Sun Microsystems Inc., 2002, see http://jcp.org/en/jsr/detail?id=101.

[De00] G. Della-Libera et. al.: Security in a Web Services World: A Proposed Architecture and Roadmap, IBM Corporation and Microsoft Corporation, 2000, see http://www-106.ibm.com/developerworks/library/ws-secmap.