Introduction to Web Services

by Hartwig Gunzer, Sales Engineer, Borland

March 2002

Table of Contents

Preface	1
The past	2
The present	2
The future: Web Services	4
SOAP	5
WSDL	9
UDDI	14
Conclusion	16
References	17

Preface

Everybody is talking about Web Services, and almost every company creates another definition for the term, though all are similar. This paper starts with the history of distributed programming to show that Web Services are not a reinvention of the wheel but an evolutionary step.

Afterwards, an overview of Web Services as well as a formal definition is presented. Therefore, the definition of Web Services from WebServices.org is discussed instead of creating yet another definition. The rest of the paper explains the three main technologies used in connection with Web Services: SOAP, WSDL, and UDDI.

Throughout the chapters these technologies are applied to a single example, which is a function that returns a ticker symbol for a given company name.

This paper is useful for managers needing a broad presentation of Web Services, and it is useful for developers who need technical details. Since SOAP, WSDL, and UDDI are based on XML, the reader should have some basic knowledge about XML, XML schemas, and XML namespaces. The references section at the end of the paper shows where to find this information.

Web Services

white paper

The past

With the emergence of computer networks, the paradigm of distributed computing was born. Applications were split first into two parts with one part, the client, initiating a distributed activity, and the other part, the server, carrying out that activity. This decentralization minimized bottlenecks by distributing the workload across multiple systems. It provided flexibility to application design formerly unknown on centralized hosts. But this two-tier architecture had its limits. For failover and scalability, issues a third tier was introduced, separating an application into a presentation part, a middle tier containing the business logic, and a third tier dealing with the data. This three-tier model of distribution has become the most popular way of splitting applications. It makes application systems scalable. The foundation for the communication between the distributed parts of an application is the remote procedure call (RPC). To keep developers from the low-level tasks like data conversion or the byte order of different machines, a new layer of software hit the market. This middleware masks the differences between various kinds of hosts. It sits on top of the host's operating system and networking services and offers its services to the applications above.

The first middlewares, like DCE, were based on a procedural programming model and were superseded by the introduction of the object oriented programming model by middlewares like CORBA,® DCOM, or RMI, which are the most popular middlewares at present

The present

As we have discussed, there are basically three middlewares you can choose from: DCOM, CORBA, and RMI. All of them have their advantages as well as their disadvantages. This chapter discusses the main features without going too much into detail. It will point out how Web Services fit into the picture.

CORBA®

CORBA is an open standards-based solution to distributed computing. The Object Management Group (OMG™), an industry consortium, developed the specification for CORBA and specified the Internet InterORB Protocol (IIOP®), the standard communication protocol between Object Request Brokers (ORB™).

The primary advantage of CORBA is that clients and servers can be written in any programming language. This is possible because objects are defined with a high level of abstraction provided by the Interface Definition Language (IDL™). After defining an IDL file, a compiler takes care of the mapping of the IDL file to a specific programming language. The communication between objects, clients, and servers are handled through ORBs. If you need high performance in a distributed application, then CORBA is still the middleware of choice. However, writing a distributed application with CORBA is a fairly complex issue. In order to enable IDL to be translated into various languages, it has to limit itself to concepts that are found in all the supported languages, thus it represents a least common denominator.

In order to cover many fields of applications, CORBA provides various so-called services. This makes the specification very heavy and adds to the complexity. In addition, vendors have implemented only a small subset of services so far.

In order to make the client and the server object communicate, compatible ORBs are needed on both sides of the connection.

RMI

Remote Method Invocation (RMI) enables you to create distributed Java™-to-Java applications, in which the methods of remote Java objects can be invoked from other Java™ virtual machines(JVM™), possibly on different hosts. A Java program can make a call on a remote object once it obtains a reference to the remote object, either by looking up the remote object in the bootstrap naming service provided by RMI or by receiving the reference as an argument or a return value. A client can call a remote object in a server. That server can also be a client of other remote objects. RMI uses object serialization to marshal and unmarshal parameters and does not truncate types, supporting true object-oriented polymorphism. The protocol used for the communication is the Java Remote Method Protocol (JRMP).

Programming with RMI is straightforward once the developer has gained some experience with the Java programming language and distributed applications. No abstract language like IDL is used to describe the remote server object. In addition, RMI supports distributed garbage collection.

On the other hand, you can only use it with Java on both sides of the connection. And the ease of use is achieved by keeping the middleware fairly simple. It doesn't provide any services as those specified by CORBA. These issues have to be addressed by the developer.

DCOM

The Microsoft® Distributed Component Object Model (DCOM) allows calls to remote objects by using a layer that sits on top of the DCE RPC mechanism and which interacts with the COM runtime services. A DCOM server publishes its methods to the clients by supporting multiple interfaces. These are written in IDL, which is similar to C++. The IDL Compiler creates proxies/stubs and skeletons similar to a CORBA IDL compiler, but registers them also in the system's registry. In addition, type libraries are created. These are files that describe the remote object and which can be queried by interfaces provided by the COM mechanism. The protocol used here is the Object Remote Procedure Call (ORPC).

The binary level specification of DCOM allows the usage of various languages to code the server objects.

Like RMI, DCOM supports distributed garbage collection of remote server objects. This is achieved by a pinging mechanism that checks if the clients are still alive. On the server side, a reference counter for the clients is maintained. If it reaches zero, the object will be released.

Most people associate DCOM with Microsoft operating systems. However, ports exist for Unix as well as VMS platforms and Apple® Macintosh.®

A first conclusion

As seen in the previous section, all the three middlewares work in a similar way. The differences are found more in the different features that are supported as well as the level of complexity. All result in a tight coupling of the client and the server as you can use any one of the middlewares. Because of the different protocols, you can't call a DCOM server from a RMI client. (One step to solve this problem is a calling mechanism called RMI over IIOP, which is used in connection with Enterprise JavaBeans™ development). The connection is established on a point-to-point basis.

In addition, these middlewares are typically used for intranet applications, and it is quite an issue to cross a firewall. All of them provide HTTP tunneling to get to the server behind the firewall.

The future: Web Services

This section describes the Web Services and how they are related to traditional middlewares. After an overview, we will present the three fundamental parts that Web Services consist of: SOAP, WSDL, and UDDI.

Overview

Roughly speaking, Web Services are applications that can be published, located, and invoked across the Internet. Typical examples include

- Getting stock price information.
- Obtaining weather reports.
- Making flight reservations.

These Web Services may use other Web Services in order to perform their task.

So far, there's no difference between a Web Service and a server in a distributed application. The differences lie in the underlying layers for performing the application logic and data manipulation.

The points mentioned in the conclusion of the last part are some of the main reasons why those middlewares don't qualify for Web Services. On the worldwide Web, there are heterogeneous environments on both the server and the client side, and it is not known in advance which kind of middleware each side of the connection uses. Thus, a new approach is needed for Web Services.

Definition

To have a formal background, we state the definition given by WebServices.org:

"Web Services are encapsulated, loosely coupled contracted functions offered via standard protocols" where:

- "Encapsulated" means the implementation of the function is never seen from the outside.
- "Loosely coupled" means changing the implementation of one function does not require change of the invoking function.
- "Contracted" means there are publicly available descriptions of the function's behavior, how to bind to the function as well as its input and output parameters.

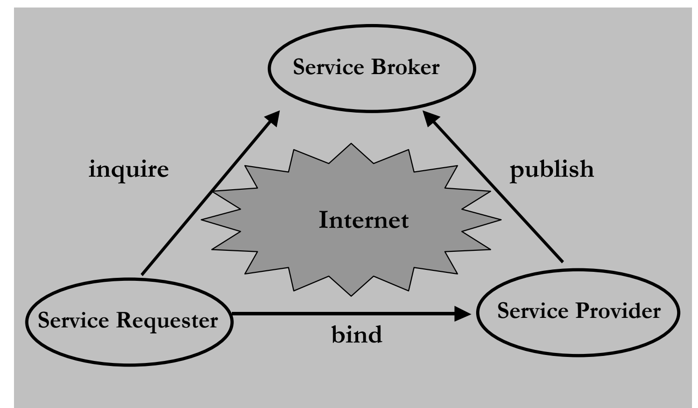
Architecture

In the traditional client server world, we had the server offering some functionality that could be used or called by the client. Some kind of look up service acted as a broker between the client and the server.

Since Web Services represent just another paradigm for distributed applications, they consist of the same three components:

- A service broker that acts as a look up service between a service provider and a service requestor.
- A service provider that publishes its services to the service broker.
- A service requester that asks the service broker where to find a suitable service provider and that binds itself to the provider.

The following drawing illustrates the relationships between the Web Services components:

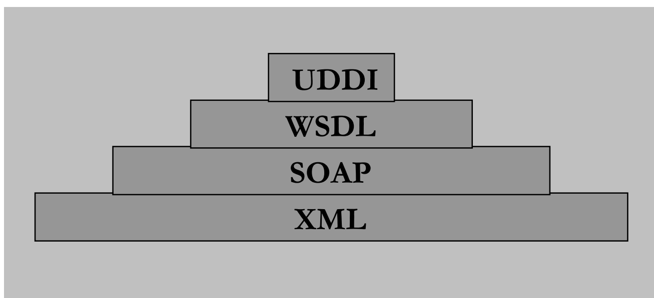


The middlewares discussed so far used some kind of a binary protocol for communication. Web Services however use XML on top of HTTP. Thus, no problems with firewalls will occur. Usually the firewalls don't block the HTTP port. If you look at the definition, you will see that Web Services don't have to use HTTP. The use of SMTP or FTP can be considered.

XML is a widely accepted format for exchanging data and its corresponding semantics. It is a fundamental building block for nearly every other layer that is used for Web Services.

Altogether, these layers build the so-called Web Services stack that consists of the following parts:

- XML (Extensible Markup Language).
- SOAP (Simple Object Access Protocol).
- WSDL (Web Services Definition Language).
- UDDI (Universal Discovery Description Integration).



The picture above is by no means complete. The layers shown build a foundation to enable the development of Web Services consisting of the three service components in accordance with the formal Web Services definition.

The following sections will explain the layers on top of XML in detail.

SOAP

Since Web Services will run in a heterogeneous environment, the protocols used to perform the data transfer between functions have to be independent of any runtime environment. SOAP is a protocol having these characteristics.

SOAP does not itself define any application semantics such as a programming model or implementation specific semantics, i.e., distributed garbage collection. It rather defines a simple mechanism for expressing application semantics by providing a modular packaging model and encoding mechanisms for encoding data within modules.

SOAP messages contain the tag ENVELOPE as the root element. The envelope contains two elements:

- An optional header.
- A body.

The header may contain a set of so-called header entries that can be used to provide information such as authentication or the encoding of the data. In addition, two header attributes exist to provide information about how a recipient of a SOAP message should process the message:

- The actor attribute.
  This attribute can be used to specify the recipient of a certain header entry. This might not be another recipient than the final one, if the SOAP message passes some intermediate recipients on its way to the final recipient. If the message contains header entries that are enclosed for such an intermediate recipient, this recipient is not allowed to pass this header entry on to any successors. However, it is not forbidden to add any further header entries.

- The mustUnderstand attribute.
  By adding this attribute to a header entry with a value of "1" the recipient can be forced to process this header entry. Thus, it can be ensured that this entry will not be ignored.
  Not specifying this attribute has the same semantics as specifying it with a value of "0".

All header entries have to be namespace qualified.
In addition, namespaces for the envelope and the encodingStyle are defined within the SOAP specification.

The body contains information for the last recipient in the order of all recipients. This information is encapsulated in body entries. If the envelope contains a header, the body must not be the first element within the envelope. The body entries may be namespace qualified.

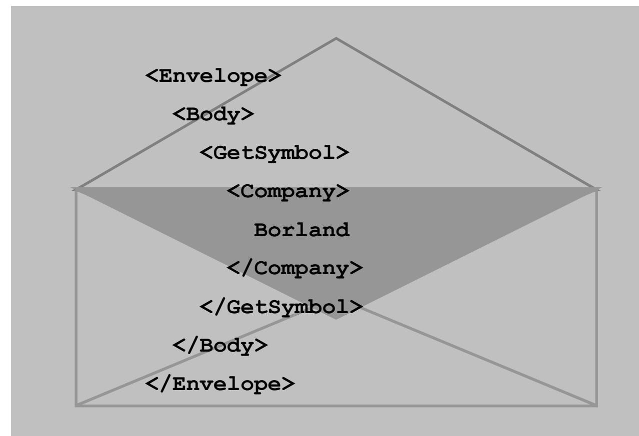
The specification defines one body entry that is used to carry error or status information concerning the message. It may appear only once in a SOAP message. Any further description of the fault element will be given at the end of this chapter.

Throughout this chapter, we will show the usage of SOAP for one particular function. In Java, it would look like this:

```
public String getSymbol ( String company );
```

This function takes one parameter, which denotes a company name. It returns a string, which represents the symbol used for getting stock quotes.

The following picture shows the elements a SOAP message must at least contain to describe our function:



Though this example was built according to the specification, it doesn't contain enough information to act as an RPC call, e.g., the type of the parameter.

Encapsulating RPCs is an important aspect where SOAP is used. For this purpose, the specification defines a global attribute called "encodingStyle" which is used for serialization of data and thus can be used for data marshalling during RPC calls. It can appear on any element. Child elements without their own encodingStyle attribute are in the scope of the encodingStyle attribute of the father element that is closest in the hierarchy. Though the data encoding rules defined by XML Schema Specification would be sufficient, SOAP uses a subset of these rules.

Like programming languages, SOAP specifies primitive data types like short, int, float, but also strings, enumerations, arrays and compound data types. Each of the parts of the compound types contains a type as well. The types are adopted from the XML Schema Specification (please refer to the references section).

RPC calls are encoded within the body of a message. The pieces of information needed are:
To make a method call, the following information is needed:

- The URI of the target object.
- A method name.
- An optional method signature.
- The parameters to the method.
- Optional header data.

The URI of the target object is not specified within the envelope of the message. This issue is deferred to the protocol binding. If we use HTTP as the underlying protocol, the HTTP request URI defines the target.

Each SOAP method contains only a single method call.
A call is described by a compound data type having the same name as the method. There is an "accessor" for each parameter with the same name and type as the parameter that is to specified. The order of the parameters has to be the same as in the specified function.

Thus, our simple example would have to look like this to serve as a RPC. We enhance it by using namespaces.

```
<SOAP-ENV:Envelope

 xmlns:SOAP-ENV=
"http://schemas.xmlsoap.org/soap/envelope/"

 SOAP-ENV:encodingStyle=
"http://schemas.xmlsoap.org/soap/encoding/">

  <SOAP-ENV:Body>
    <mtd:GetSymbol xmlns:mtd="an URI"
      <company xsi:type="xsd:string">
        Borland
      </company>
    </mtd:GetSymbol>
  </SOAP-ENV:Body>

</SOAP-ENV:Envelope>
```

As you can see, SOAP specifies basically a one way protocol.

In order to establish a request/response communication a second SOAP message will be send from the provider to the requester.

These two messages have no obvious connection to each other, that is, SOAP doesn't define how the names of the accessors have to look. There has to be an accessor for the return value as well as accessors for each out and each in/out parameter. The order has to be the same as in the request message with the accessor for the return value being the first one.

In addition, a coding convention is introduced to name the response struct after the request struct followed by the string "Response".

The next drawing shows a response to the request we have just sent.

```
<SOAP-ENV:Envelope

 xmlns:SOAP-ENV=
"http://schemas.xmlsoap.org/soap/envelope/"

 SOAP-ENV:encodingStyle=
"http://schemas.xmlsoap.org/soap/encoding/">

  <SOAP-ENV:Body>
    <mtd:GetSymbolResponse
        xmlns:mtd="an URI">
      <return xsi:type="xsd:string">
        BORL
      </return>
    </mtd:GetSymbolResponse>
  </SOAP-ENV:Body>

</SOAP-ENV:Envelope>
```

What we have not mentioned so far is the binding of SOAP to an underlying transport protocol. For HTTP, there is a mandatory header field called SOAPAction. This field can be used to indicate the intent of the SOAP HTTP request. The value is a URI identifying the intent. SOAP places no restrictions on the format or specificity of the URI or that it is resolvable. An HTTP client must use this header field when issuing a SOAP

HTTP Request. This field can be processed by servers or firewalls to filter the message accordingly.

So finally, we present the complete SOAP example for issuing a RPC call via HTTP:

```
POST /Symbol HTTP/1.1
Host: www.borland.com
Content-Type: text/xml; charset="utf-8"
Content-Length: nnnn
SOAPAction: "www.borland.com/Symbol"

<SOAP-ENV:Envelope

 xmlns:SOAP-ENV=
"http://schemas.xmlsoap.org/soap/envelope/"

 SOAP-ENV:encodingStyle=
"http://schemas.xmlsoap.org/soap/encoding/">

  <SOAP-ENV:Body>
    <mtd:GetSymbol
      xmlns:mtd="an URI">
      <return xsi:type="xsd:string">
        BORL
      </return>
    </mtd:GetSymbol>
  </SOAP-ENV:Body>
```

If the request fails for some reason, the body's fault entry will be used to explain the failure. The SOAP fault element contains four sub-elements:

- faultcode
  The specification defines four codes to describe the type of failure:
  1. VersionMismatch:
     The message did not contain the namespace http://schemas.xmlsoap.org/soap/envelope/
  2. MustUnderstand:
     An element with the mustUnderstand Attribute set to "1" was not processed correctly.
  3. Client:
     The message was incorrectly formed.
  4. Server:
     The message could not be processed correctly, but the error did not belong to the contents of the message itself.
- faultstring
  Provides a readable explanation.
- faultactor
  Specifies the entity that caused the failure on the message's way to its final destination.
- detail
  Carries application specific error information.

We will show a complete SOAP message with a possible fault entry for our example in case no symbol could be found:

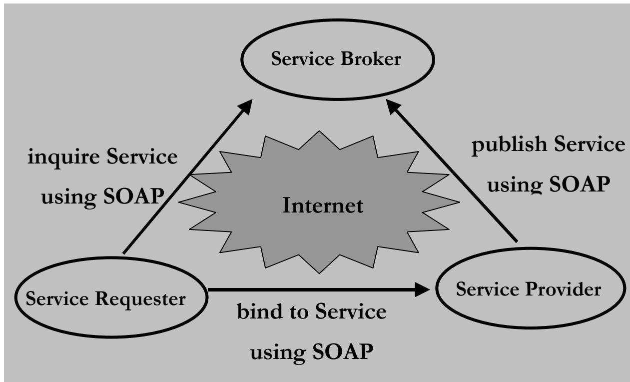
```
HTTP/1.1 500 Internal Server Error
Host: www.borland.com
Content-Type: text/xml; charset="utf-8"
Content-Length: nnnn

<SOAP-ENV:Envelope

 xmlns:SOAP-ENV=
"http://schemas.xmlsoap.org/soap/envelope/"

 <SOAP-ENV:Body>
   <SOAP-ENV:Fault>
     <faultcode>
        SOAP-ENV:Server
     </faultcode>
     <faultstring>
        Server Error
     </faultstring>
     <detail>
        <d:message xmlns:d="some URI">
          No Symbol found
        </d:message>
     </detail>
   </SOAP-ENV:Fault>
 </SOAP-ENV:Body>
```

The next image shows the way SOAP is used in our Web Services picture:



Now that you have acquired some understanding of SOAP, we will explain how to describe the features of a Web Service.

WSDL

WSDL describes network services by using an XML grammar. It provides documentation for distributed systems and has the goal to enable applications to communicate with each other in an automated way.

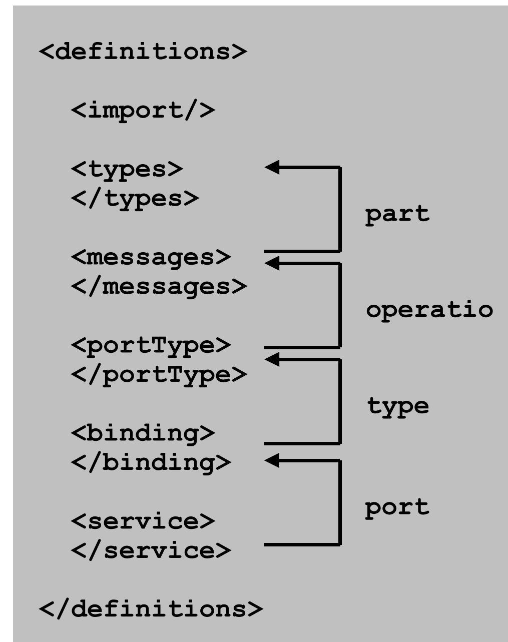
While SOAP specifies the communication between a requester and a provider, WSDL describes the services offered by the provider (an "endpoint") and might be used as a recipe to generate the proper SOAP messages to access the services.
A WSDL document has a role similar to an IDL file in CORBA or the Remote Interface in a Java RMI implementation.
The document structure and its elements are described in the following chapters.

Document structure

The root element of any WSDL document is the element <definitions>. It contains six elements that can be divided into two groups: abstract definitions and concrete definitions:

- Abstract definitions:
  1. types
  2. messages
  3. portType
- Concrete definitions:
  1. bindings
  2. services

The elements contain references to each other as shown in the following picture. The description next to the arrow represents the name of the element's attribute that contains the reference.



Each element within the document may contain a <documentation> element to provide some human readable explanation of the element that is specified. In addition, all the elements contain an attribute "name" serving as an identifier. Though in the context of Web Services WSDL is used to describe how to build SOAP requests and what kind of SOAP responses to expect, it is designed to support other bindings as well.

The name attribute of this element specifies the name of the Web Service. Usually it corresponds to the file name. Next to the elements already mentioned, it contains a set of namespace

definitions that are used throughout the document. Example namespaces used in a WSDL document are the following ones:

```
<?xml version="1.0"?>
 <definitions name="StockQuote"
   targetNamespace=
  "http://borland.com/symbol/definitions"

xmlns:tns="http://borland.com/symbol/definitions"
xmlns:xsd1="http://borland.com/symbol/schemas"
xmlns:soap="http://schemas.xmlsoap.org/wsdl/soap/"
xmlns="http://schemas.xmlsoap.org/wsdl/">

</definitions>
```

The targetNamespace attribute declares a namespace to which all names declared in the document will belong.

The import element

By using the import statement, you can separate a WSDL document into multiple independent documents, thus keeping the structure clearer and easier to maintain. One way of separating the different parts could be the level of abstraction each part belongs to.

Afterwards the elements can be imported as needed.
The syntax for the import element is:

```
<import namespace="an uri" location="an uri"/>
```

Each document may have multiple import statements.

The types element

All data type definitions relevant for sending and receiving the messages described are contained within the types element. The specification suggests using XML schema definitions.
To describe our example from above in WSDL, we start with the following type definition:

```
<types>
  <schema
    targetNamespace="http://borland.com/symbol.xsd"
    xmlns="http://www.w3.org/2000/10/XMLSchema">
      <element name="SymbolResponseType">
        <complexType>
        <all>
         <element name="tickerSymbol" type="string"/>
        </all>
        </complexType>
      </element>
      <element name="SymbolRequestType">
        <complexType>
          <all>
           <element name="company" type="string"/>
          </all>
        </complexType>
    </element>
  </schema>
</types>
```

Throughout the WSDL document we will use these two type definitions called “tickerSymbol” and “companyName”, both representing a string.

The message element

Now that we have defined the types we want to use, we have to specifiy the data that will be communicated between a service requester and a service provider.

Therefore the specification defines the message element. A message element may appear multiple times and consists of a name and one or more <part> elements. The part elements refer to the already defined types using a type attribute. In other words, the parts specify the contents of the message.

Since our example uses one input parameter and a single return value, we specify the following messages. Please note that so far this is just an abstract definition of a data flow. Nothing tells us that these two messages belong to the same operation.

```
<message name="GetSymbolInput">
      <part name="inputparam"
      element="xsd1:SymbolRequestType"/>
 </message>
 <message name="GetSymbolOutput">
      <part name="returnvalue"
      element="xsd1:SymbolResponseType"/>
 </message>
```

The port type element

A port type is a set of related operations. Therefore, an element <operation> is introduced, having a name attribute and another optional attribute specifying the order of parameters used in this operation. In addition, to identify the kind of operation, four so-called transmission primitives can be found in the specification:

- one-way:
  The endpoint receives a message.
- request/response:
  The endpoint receives a message and sends a reply.
- solicit response
  The endpoint sends a message and receives a reply.
- notification
  The endpoint sends a message.

These primitives are described by using three sub elements within the operation element:

- <input ... / >
- <output ... / >
- <fault ... / >

All these elements contain a "name" attribute and a "message" attribute referring to a previously defined message element. The following rules apply:

- A one-way operation only specifies the input element.
- A request/response operation specifies the input element first, followed be the output element and an some optional fault elements.
- A solicit response operation specifies the output element first, then the input element followed by some optional fault elements.
- A notification operation only specifies the output element.

Using a portType, we can now define a "GetSymbol" operation that has the "GetSymbolInput" message as an input and that produces the "GetSymbolOutput" message as an output. Thus, we define a request/response operation:

```
<portType name="SymbolPortType">
  <operation name="GetSymbol">
    <input message="tns:GetSymbolInput"/>
    <output message="tns:GetSymbolOutput"/>
  </operation>
</portType>
```

We haven't specified the name attribute for the input and the output element. According to the specification, they will have a default value of the name of the operation appended with "Request" and "Response" respectively.

The binding element

The elements presented so far described our operation in a generic way. Nothing has been said about a concrete implementation. So we could even think of a CORBA implementation. Our next goal is to bind the operation to the SOAP protocol.

To serve this purpose, the WSDL specification introduces the <binding> element. Next to a name attribute it contains a type attribute that references a portType and provides predefined binding for protocols such as SOAP, HTTP, and MIME. The grammar looks similar to the portType grammar. There has to be at least one binding for every portType.

Once more, there is an operation element having three sub-elements: input, output, and fault. Each operation corresponds to an operation in the portType element.

Binding to SOAP

To bind the portType to SOAP, there is a SOAP extension element called <soap:binding>. It provides two parameters: the transport protocol and the request style, which can be either "rpc" or "document". This value serves as a default value within the SOAP operations.

Since we want to use SOAP for sending an RPC request over HTTP we specify:

```
<soap:binding style="rpc"
transport="http://schemas.xmlsoap.org/soap/http"/>
```

Now we have to provide information for the operation. We have to use the <soap:operation> element. It contains an attribute "style" to override the default style for this particular operation and an attribute "soapAction" that is used for the HTTP header of a SOAP message. (Please refer to the chapter about SOAP.) To be consistent with our example, we have to write:

```
<soap:operation style = "rpc"
soapAction="www.borland.com/Symbol"/>
```

By using the "rpc" style, each message part will appear in a wrapper element containing the name of the operation. This is exactly what we need for our SOAP message.

To state the way a message shall appear in the body of a SOAP message a <soap:body> element is defined within the WSDL specification. It appears either in the <input> or the <output> element of the binding.

The SOAP body contains four elements:

- parts
  This optional attribute can be used to tell which part has to appear where in the message. If it is left out all parts described will be included in the SOAP body.
- use
  This required attribute states if the message parts are encoded using an encoding style (value "encoded") or whether a part references a concrete schema definition(value "literal").
- encodingStyle
  This attribute has exactly the same semantics as in the SOAP specification.
- namespace
  We may define another namespace to avoid name clashing.

A <soap:fault> element exists to specify a SOAP fault entry. It has the same pattern as the soap:body element.

To define header entries and header fault elements the elements <soap:header> and <soap:headerfault> are used the same way as the SOAP elements mentioned above. In addition the soap:header contains two attributes, "message" and "part". They refer to the message part in the WSDL document that defines the header type.

We present the complete binding element for our example:

```
<binding name="SymbolSoapBinding"
    type="tns:SymbolPortType">
    <soap:binding style="rpc"
        transport=
        "http://schemas.xmlsoap.org/soap/http"/>
    <operation name="GetSymbol">
      <soap:operation style = "rpc"
        soapAction="www.borland.com/Symbol"/>
      <input>
          <soap:body
            use="encoded"
            encodingStyle=

"http://schemas.xmlsoap.org/soap/encoding/" />
      </input>
     <output>
          <soap:body
            use="encoded"
            encodingStyle=

"http://schemas.xmlsoap.org/soap/encoding/" />
      </output>
   </operation>
</binding>
```

The service element

A service is a set of related ports. That is why the service element only contains a name attribute and a sub-element <port> that may appear multiple times.

Different ports are used if multiple implementations of the same binding exist.

The binding is specified within the port element.
In order to bind to the SOAP protocol a <soap:address> is defined. It contains a name and a location that defines an address of the same type as the transport specified in the soap:binding element. That means that we have to define a HTTP address.

We finish this section by showing the service element for our example:

```
<service name="SymbolService">
     <documentation>a simple web service
     </documentation>
     <port name="SymbolPort"
       binding="tns:SymbolSoapBinding">
       <soap:address
          location="http://borland.com/symbol"/>
     </port>
</service>
```

UDDI

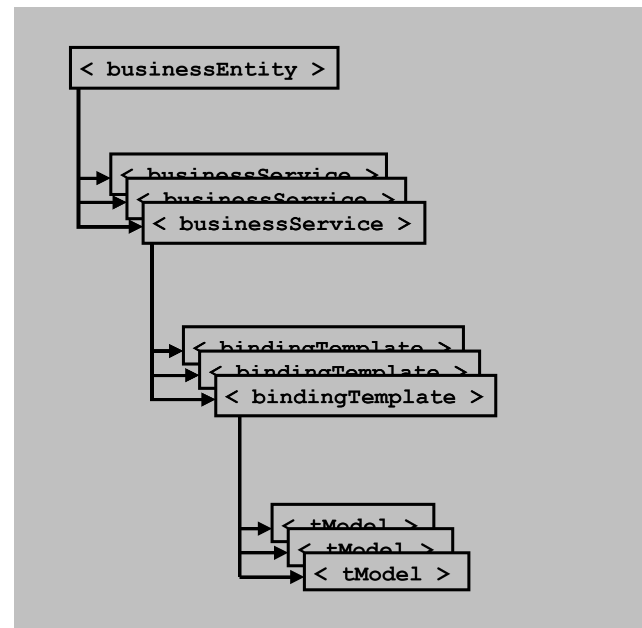
Universal Description, Discovery, and Integration (UDDI) is a standard designed to provide a searchable directory of businesses and their Web Services. Thus, it represents the service broker that enables service requesters to find a suitable service provider. In many ways UDDI is designed like a phone book. It contains support for:

- yellow pages taxonomies
  Searches can be performed to locate businesses which service a particular industry or product category, or that are located within a specific geographic region.
- white pages
  Information about a service provider, including address, contact, and known identifiers.
- green pages
  Technical information about Web Services that are exposed by the business, e.g., how to communicate with the Web Service.

To get access to the UDDI services, the UDDI directory exposes a set of APIs in the form of a SOAP-based Web Service. The API is divided into two logical parts. These are the Inquiry API and the Publishers' API. The Inquiry API consists of two further parts—one part used for constructing programs that let you search and browse information found in a UDDI registry, and another part that is used in case of invocation failures.
The core component is the UDDI business registration, an XML file used to describe a business entity and its Web Services.
The specification defines a SOAP-based programming protocol based on HTTP for registering and discovering Web Services. Using the UDDI discovery services, businesses individually register information about the Web Services that they expose for use by other businesses. This information can be added to the UDDI business registry either via a Web site or by using tools that make use of the programmatic service interfaces described in the UDDI Programmer's API Specification.

The four key data structures

The UDDI XML schema defines four core types of information needed by developers in order to bind to Web Services.
The following image shows the relationships between the different structures:



As shown above the datastructures consist of:

- business entities
  This data structure contains information about the company that publishes the Web Service. This information includes at least the name of the company, but may also give information about discovery URLs, the services offered, as well as how to contact a person from that company (address, phone number, email, etc.).
  Each business entity may contain multiple business services.
- business services
  Use this data structure to describe each offered service in business terms. You have to specify at least a name and one binding template.

Each business service may contain multiple binding templates, but at least one.

- binding templates
  Each binding templates structure serves as a container for one or more binding template structures. A binding template structure describes how to get access to a service. These so-called access points represent URLs. Valid values for the accessPoint element are:
  1. mailto
  2. http
  3. https
  4. ftp
  5. fax
  6. phone
  7. other

  Another attribute called <hostingRedirector> can be used if the binding template refers to another one that has already been specified. So in this case, the access point doesn't have to be specified a second time.

  Another element called <tModelInstanceDetails> holds information for a particular tModel, which is referenced by its key. We described tModels in the next section.

- tModels
  The tModels are used to describe a technical specification, e.g. wire protocols, interchange formats, or sequencing rules. They enable parties to find out, if they are compatible. If further parties want to express compliance with a specification they just have to provide a reference to the appropriate tModel.

  There exist two mandatory attributes within a tModel:
  1. tModelKey
     This attribute serves as a unique identifier among all the tModels. If you want to update an existing tModel you have to provide the key of that model.
  2. name
     To provide a meaningful name for the tModel this attribute has to be specified.

  The other attributes provide support for the API used to search a UDDI registry as well as some further documentation. An optional, though important, attribute is the <overviewDoc>. This structure points via a URL to a remote document containing descriptive information or instructions.

```
<businessService businessKey="..."
  serviceKey="...">
  <name> SymbolService </name>
  <description> a description </description>
  <bindingTemplates>
    <bindingTemplate>
      ...
      <accessPoint urlType="http">
        http://borland.com/symbol
      </accessPoint>
      <tModelInstanceDetails>
        <tModelInstanceInfo
          tModelKey="123123">
        </tModelInstanceInfo>
      </tModelInstanceDetails>
    </bindingTemplate>
  </bindingTemplates>
</businessService>
```

We will now connect UDDI to our Web Services picture by showing how to register WSDL documents in UDDI.

Using WSDL definitions in UDDI

Each WSDL document will be specified in a tModel element. Each such tModel must be classified, as being of type "wsdlSpec" and must have an overviewDoc pointing to the desired WSDL document.

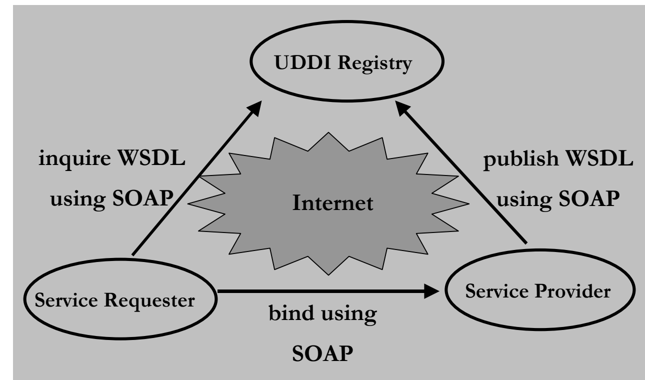
The types used belong to the uddi-org:types taxonomy.
We will now use UDDI with our example from the previous chapters.

We start with the tModel containing an overviewURL pointing to the WSDL file:

```
<tModel
  authorizedName="..." operator="..."
  tModelKey="123123">
  <name> StockQuote Service </name>
  <description xml:lan="en">
    WSDL description of a Symbol lookup service
  </description>
  <overviewDoc>
    <description xml:lang="en">
      WSDL source document.
    </description>
    <overviewURL>
      http://www.borland.com/symbol.wsdl
    </overviewURL>
  </overviewDoc>
  <categoryBag>
    <keyedReference
      tModelKey="UUID:123123"
      keyName="uddi-org:types"
      keyValue="wsdlSpec"/>
  </categoryBag>
</tModel>
```

Conclusion

Having described the key technologies, we now present the complete picture of distributed applications as it applies to SOAP, WSDL, and UDDI:



1. The Web Service provider describes the Web Service in a WSDL document and publishes it to a UDDI registration using the Publisher's API (which is based on SOAP).
2. A service requester uses the UDDI Inquiry API to search the UDDI registration for a suitable service provider. If one is found, it can search for the tModel pointing to the WSDL document.
3. A SOAP request will be created according to the WSDL document.
4. The SOAP request will be sent to the service provider, and the response will be processed.

Though creating all the necessary documents seems complicated, please note that they are usually generated similar to stubs and skeletons in other middlewares. The developer doesn't have to care about the communication issues. Toolkits and APIs will generate automatically the appropriate XML documents.

Throughout this paper, Web Services were seen as "yet another middleware," similar to other middleware such as CORBA and RMI, but definitely more suitable to be used across the Internet.

Thus, typical issues concerning distributed application programming like performance, scalability, failover and security have to be faced also when dealing with Web Services. Just imagine if your own Web Service uses another Web Service for credit card authentication. What happens to your Web Service or even to your whole business if the security service is not available? Or how should your Web Service react if it experiences a kind of rush hour with thousands of client requests? These issues are the next steps Web Services have to take in order to become a mature platform. However, the first promising steps have been made.

References

W3C XML Specification

http://www.w3.org/TR/REC-xml

W3C XML Schema Specification

http://www.w3.org/TR/xmlschema-1/

http://www.w3.org/TR/xmlschema-2/

W3C XML Namespaces Specification

http://www.w3.org/TR/REC-xml-names/

W3C SOAP Specification

http://www.w3.org/TR/SOAP

W3C WSDL Specification

http://www.w3.org/TR/WSDL

UDDI Specifications

http://www.uddi.org/specifications

WebServices.org Homepage

http://www.webservices.org

Borland®

100 Enterprise Way
Scotts Valley, CA 95066-3249
www.borland.com | 831-431-1000

Made in Borland® Copyright © 2002 Borland Software Corporation. All rights reserved. All Borland brand and product names are trademarks or registered trademarks of Borland Software Corporation in the United States and other countries. All other marks are the property of their respective owners. 12728