Sponsored by:

JAVAWORLD
SOLUTIONS FOR JAVA DEVELOPERS

This story appeared on JavaWorld at
http://www.javaworld.com/javaworld/jw-03-2001/jw-0330-soap.html

Clean up your wire protocol with SOAP, Part 1

An introduction to SOAP basics

By Tarak Modi, JavaWorld.com, 03/30/01

Many developers have run into this dilemma: A CORBA client needs to obtain the services of a Distributed Component Object Model (DCOM) client or vice versa. The common solution is to use a COM/CORBA bridge, however, this answer is fraught with failure points. Suppose you have just introduced a complex new piece of software in the midst of two already complicated pieces (the CORBA ORB and the COM infrastructure). The bridge's complexity results from the intricate back-and-forth translation that it must complete from CORBA's Internet Inter-ORB Protocol (IIOP) to DCOM's Object Remote Procedure Call (ORPC). Any changes to these protocols mean changes to the bridge. What if I tell you that SOAP can alleviate the problem? Interested?

SOAP stands for Simple Object Access Protocol. In a nutshell, SOAP is a wire protocol similar to the IIOP for CORBA, ORPC for DCOM, or Java Remote Method Protocol (JRMP) for Java Remote Method Invocation (RMI). At this point you may be wondering, with so many wire protocols in existence, why do we need another one. In fact, isn't that what caused the problem discussed in the opening paragraph in the first place? Those are valid questions, however SOAP is somewhat different from the other wire protocols.

Let's examine how:

- While IIOP, ORPC, and JRMP are binary protocols, SOAP is a text-based protocol that uses XML. Using XML for data encoding gives SOAP some unique capabilities. For example, it is much easier to debug applications based on SOAP because it is much easier to read XML than a binary stream. And since all the information in SOAP is in text form, SOAP is much more firewall-friendly than IIOP, ORPC, or JRMP.
- Because it is based on a vendor-agnostic technology, namely XML, HTTP, and Simple Mail Transfer Protocol (SMTP), SOAP appeals to all vendors. For example, Microsoft is committed to SOAP, as are a variety of CORBA ORB vendors such as Iona. IBM, which played a major role in the specification of SOAP, has also created an excellent SOAP toolkit for Java programmers. The company has donated that toolkit to Apache Software Foundation's XML Project, which has created the Apache-SOAP implementation based on the toolkit. The implementation is freely available under the Apache license. Returning to the problem stated in the opening paragraph, if DCOM uses SOAP and the ORB vendor uses SOAP, then the problem of COM/CORBA interoperability becomes significantly smaller.

SOAP is not just another buzzword; it's a technology that will be deeply embedded in the future of distributed computing. Coupled with other technologies such as Universal Discovery, Description, and Integration (UDDI) and Web Services Description Language (WSDL), SOAP is set to transform the way business applications communicate over the Web with the notion of Web services. I can't emphasize enough the importance of having the knowledge of SOAP in your developer's toolkit. In Part 1 of this four-part series on SOAP, I will cover the basics, starting with how the idea of SOAP was conceived.

Read the whole series on SOAP:

- Part 1: An introduction to SOAP basics
- Part 2: Use Apache SOAP to create SOAP-based applications
- Part 3: Create SOAP services in Apache SOAP with JavaScript

- Part 4: Dynamic proxies make Apache SOAP client development easy

Inside SOAP

As I mentioned above, SOAP uses XML as the data-encoding format. The idea of using XML is not original to SOAP and is actually quite intuitive. XML-RPC and ebXML use XML as well. See Resources for references to Websites where you can find more information.

Consider the following Java interface:

Listing 1

```
public interface Hello
{
    public String sayHelloTo(String name);
}
```

A client calling the `sayHelloTo()` method with a name would expect to receive a personalized "Hello" message from the server. Now imagine that RMI, CORBA, and DCOM do not exist yet and it is up to you to serialize the method call and send it to the remote machine. Almost all of you would say, "Let's use XML," and I agree. Accordingly, let's come up with a request format to send to the server. Assuming that we want to simulate the call `sayHelloTo("John")`, I propose the following:

Listing 2

```
<?xml version="1.0"?>
<Hello>
    <sayHelloTo>
        <name>John</name>
    </sayHelloTo>
</Hello>
```

I've made the interface name the root node. I've also made the method and parameter names nodes as well. Now we must deliver this request to the server. Instead of creating our own TCP/IP protocol, we'll defer to HTTP. So, the next step is to package the request into the form of an HTTP `POST` request and send it to the server. I will go into the details of what is actually required to create this HTTP `POST` request in a later section of this article. For now let's just assume that it is created. The server receives the request, decodes the XML, and sends the client a response, again in the form of XML. Assume that the response looks as follows:

Listing 3

```
<?xml version="1.0"?>
<Hello>
    <sayHelloToResponse>
        <message>Hello John, How are you?</message>
    </sayHelloToResponse>
</Hello>
```

The root node is still the interface name `Hello`. But this time, instead of just the method name, the node name, `sayHelloTo`, is the method name plus the string `Response`. The client knows which method it called, and to find the response to that method it simply looks for an element with that method name plus the string `Response`.

I have just introduced you to the roots of SOAP. Listing 4 shows how the same request is encoded in SOAP:

Listing 4

```
<SOAP-ENV:Envelope
                        xmlns:SOAP-ENV="
http://schemas.xmlsoap.org/soap/envelope/"
                        xmlns:xsi="
http://www.w3.org/1999/XMLSchema-instance"
                        xmlns:xsd="http://www.w3.org/1999/XMLSchema">
```

```
     <SOAP-ENV:Header>
     </SOAP-ENV:Header>
    <SOAP-ENV:Body>
         <ns1:sayHelloTo
                      xmlns:ns1="Hello"
                     SOAP-ENV:encodingStyle="
http://schemas.xmlsoap.org/soap/encoding/">
             <name xsi:type="xsd:string">John</name>
         </ns1:sayHelloTo>
    </SOAP-ENV:Body>
</SOAP-ENV:Envelope>
```

Looks slightly more complicated, doesn't it? Actually it's similar to what we did before with a few enhancements added in for extensibility. First, note how the SOAP document is neatly organized into an `Envelope` (the root node), a header section, and a body. The header section is used to encapsulate data that is not tied to a specific method itself, but instead provides context knowledge, such as a transaction ID and security information. The body section contains the method-specific information. In Listing 2, the homegrown XML only had a body section.

Second, note the heavy use of XML namespaces. `SOAP-ENV` maps to the namespace http://schemas.xmlsoap.org/soap/envelope/, `xsi` maps to http://www.w3.org/1999/XMLSchema-instance, and `xsd` maps to http://www.w3.org/1999/XMLSchema. Those are standard namespaces that all SOAP documents have.

Finally, in Listing 4 the interface name (i.e., `Hello`) is no longer the node name as it was in Listing 2. Rather it refers to a namespace, `ns1`. Also, along with the parameter value, the type information is also sent to the server. Note the value of the envelope's `encodingStyle` attribute. It is set to http://schemas.xmlsoap.org/soap/encoding/. That value informs the server of the encoding style used to encode -- i.e., serialize -- the method; the server requires that information to successfully deserialize the method. As far as the server is concerned, the SOAP document is completely self-describing.

The response to the preceding SOAP request would be as follows:

Listing 5

```
<SOAP-ENV:Envelope
                      xmlns:SOAP-ENV="http://schemas.xmlsoap.org/soap/envelope/"
                      xmlns:xsi="http://www.w3.org/1999/XMLSchema-instance"
                      xmlns:xsd="http://www.w3.org/1999/XMLSchema">
     <SOAP-ENV:Body>
          <ns1:sayHelloToResponse
                     xmlns:ns1="Hello"
                     SOAP-ENV:encodingStyle="http://schemas.xmlsoap.org/soap/encoding/">
               <return xsi:type="xsd:string">Hello John, How are you doing?</return>
          </ns1:sayHelloToResponse>
    </SOAP-ENV:Body>
</SOAP-ENV:Envelope>
```

Listing 5 resembles the request message in Listing 4. In the code above, the method parameters don't contain the return value -- which in this example is the personalized "Hello" message; the body does.

The document's format has tremendous flexibility built in. For example, the encoding style is not fixed but instead, specified by the client. As long as the client and server agree on this encoding style, it can be any valid XML.

Plus, separating the call context information means that the method doesn't concern itself with that information. Major application servers in the market today follow that same philosophy. Earlier, I indicated that context knowledge could include transaction and security information, but context knowledge could cover almost anything. Here's an example of a SOAP header with some transaction information:

Listing 6

```
<SOAP-ENV:Header>
     <t:Transaction xmlns:t="some-URI" SOAP-ENV:mustUnderstand="1">
          5
     </t:Transaction>
</SOAP-ENV:Header>
```

The namespace `t` maps to some application-specific URI. Here `5` is meant to be the transaction ID of which this method is a part. Note the use of the SOAP envelope's `mustUnderstand` attribute. It is set to `1`, which means that the server must either understand and honor the transaction request or must fail to process the message; the SOAP specification mandates that.

When good SOAP requests go bad

Just because you use SOAP does not mean that all your requests will succeed all the time. Things can go wrong in many places. For example, the server may not honor your request because it can't access a critical resource such as a database.

Let's return to our "Hello" example and add a silly constraint to it: "It is not valid to say hello to someone on Tuesday." So on Tuesdays, even though the request sent to the server is valid, the server will return an error response to the client. This response would be similar to the following:

Listing 7

```
<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>
               <e:myfaultdetails xmlns:e="Hello">
                 <message>
                   Sorry, my silly constraint says that I cannot say hello on Tuesday.
                 </message>
                 <errorcode>
                   1001
                 </errorcode>
               </e:myfaultdetails>
           </detail>
       </SOAP-ENV:Fault>
   </SOAP-ENV:Body>
</SOAP-ENV:Envelope>
```

Let's focus on the `Fault` element defined in the http://schemas.xmlsoap.org/soap/envelope/namespace. All SOAP servers must always return any error condition in that element, which is always a direct child of the `Body` element. Without exception, the `Fault` element must have `faultcode` and `faultstring` elements. The `faultcode` is a code that can identify problems; client-side software uses `faultcode` for algorithmic processing as the SOAP specification calls it. The SOAP specification defines a small set of fault codes that you can use. The `faultstring` on the other hand is meant for human consumption.

The code snippet in Listing 7 also shows a `detail` element. Since the error occurred while processing the SOAP message's body section, the `detail` element must be present. As you'll see later, if the error occurs while processing the header, `detail` must *not* be present. In Listing 7, the application used that element to provide a more detailed explanation of the nature of the error, namely that it was not allowed to say hello on Tuesdays. An application-specific error code is also present as well: a semioptional element called `faultfactor` that I have not shown in the error message. I call it semioptional because it must be included if the error message was sent by a server that was not the request's end-processing point, i.e., an intermediate server. SOAP does not specify any situation in which the `faultcode` element must not be included.

In Listing 7, the fault resulted from the method invocation itself, and the application processing the method caused it. Now let's take a look at another type of fault; one that generates as a result of the server not being able to process the header information. As an example, assume that all hello messages must generate in the context of a transaction. That request would look similar to this:

Listing 8

```
<SOAP-ENV:Envelope
                       xmlns:SOAP-ENV="
http://schemas.xmlsoap.org/soap/envelope/"
                       xmlns:xsi="
http://www.w3.org/1999/XMLSchema-instance"
                       xmlns:xsd="http://www.w3.org/1999/XMLSchema">
     <SOAP-ENV:Header>
```

```
        <t:Transaction xmlns:t="some-URI"
SOAP-ENV:mustUnderstand="1">
             5
        </t:Transaction>
     </SOAP-ENV:Header>
    <SOAP-ENV:Body>
        <ns1:sayHelloTo
                     xmlns:ns1="Hello"
                    SOAP-ENV:encodingStyle="
http://schemas.xmlsoap.org/soap/encoding/">
            <name xsi:type="xsd:string">Tarak</name>
        </ns1:sayHelloTo>
    </SOAP-ENV:Body>
</SOAP-ENV:Envelope>
```

The above message's header section has a `transaction` element; it specifies the transaction number that the method invocation must be part of. I say *must* because the `transaction` element uses the `mustUnderstand` attribute. As I mentioned before, the SOAP server must either honor that or fail to process the request. To make matters interesting, let's assume that the SOAP server cannot honor that and therefore fails the request. The response would be similar to this:

Listing 9

```
<SOAP-ENV:Envelope xmlns:SOAP-ENV="
http://schemas.xmlsoap.org/soap/envelope/">
   <SOAP-ENV:Body>
       <SOAP-ENV:Fault>
           <faultcode>SOAP-ENV:MustUnderstand</faultcode>
           <faultstring>SOAP Must Understand
Error</faultstring>
       </SOAP-ENV:Fault>
   </SOAP-ENV:Body>
</SOAP-ENV:Envelope>
```

The above code resembles the error message in Listing 7. But note that the `detail` element is absent. As I mentioned before, the SOAP specification states that this element must present itself if the error occurs while processing the header. In fact, the presence or absence of the `detail` element can quickly tell you if the error happened while processing the header or the body.

SOAP and HTTP

In my first example I sent the custom XML request to the server via HTTP and glossed over what was involved in doing so. Let's come back to that. How can I send a SOAP request (instead of the custom XML) over HTTP to the server? SOAP naturally follows the HTTP request/response message model, which provides SOAP request parameters in an HTTP request and SOAP response parameters in an HTTP response. In fact, SOAP 1.0 specifically designated HTTP as its transport protocol. SOAP 1.1 has loosened up a bit and, although it still works with HTTP, it also works with other protocols such as SMTP. In this series I will only discuss SOAP in the context of using it with HTTP.

Let's go back to the hello example. If we were to send the SOAP request to the server via HTTP it would look similar to the code below:

Listing 10

```
POST http://www.SmartHello.com/HelloApplication HTTP/1.0
Content-Type: text/xml; charset="utf-8"
Content-Length: 587
SOAPAction: "http://www.SmartHello.com/HelloApplication#sayHelloTo"
<SOAP-ENV:Envelope
                      xmlns:SOAP-ENV="
http://schemas.xmlsoap.org/soap/envelope/"
                      xmlns:xsi="
http://www.w3.org/1999/XMLSchema-instance"
                      xmlns:xsd="http://www.w3.org/1999/XMLSchema">
     <SOAP-ENV:Header>
     </SOAP-ENV:Header>
    <SOAP-ENV:Body>
        <ns1:sayHelloTo
```

```
                      xmlns:ns1="Hello"
                     SOAP-ENV:encodingStyle="
http://schemas.xmlsoap.org/soap/encoding/">
            <name xsi:type="xsd:string">Tarak</name>
         </ns1:sayHelloTo>
    </SOAP-ENV:Body>
</SOAP-ENV:Envelope>
```

Listing 10 features basically the same SOAP request as Listing 4 with some HTTP-specific code at the beginning. The first line indicates that this is a `POST` request that conforms to the rules per HTTP 1.1. The target for the post is http://www.SmartHello.com /HelloApplication. The next line indicates the content type, which must be text/xml when including SOAP entity bodies in HTTP messages. The content length specifies the payload length of the `POST` request.

The fourth line is SOAP-specific and mandatory. The `SOAPAction` HTTP request header field indicates the intent of the SOAP HTTP request. The value is a URI identifying the intent. SOAP places no restrictions on the format of the URI. In fact, the URI does not even have to be resolvable to an actual location.

One possible use of the `SOAPAction` field is by a firewall that looks at the field's value and makes a decision on whether to allow the request to pass through.

Once the server has processed the request, it will return a response to the client that looks like Listing 11 (assuming that there are no errors):

Listing 11

```
HTTP/1.0 200 OK
Content-Type: text/xml; charset="utf-8"
Content-Length: 615
<SOAP-ENV:Envelope
                       xmlns:SOAP-ENV="
http://schemas.xmlsoap.org/soap/envelope/"
                       xmlns:xsi="
http://www.w3.org/1999/XMLSchema-instance"
                       xmlns:xsd="http://www.w3.org/1999/XMLSchema">
     <SOAP-ENV:Body>
          <ns1:sayHelloToResponse
                     xmlns:ns1="Hello"
                     SOAP-ENV:encodingStyle="
http://schemas.xmlsoap.org/soap/encoding/">
               <return xsi:type="xsd:string">Hello John, How are
you doing?</return>
          </ns1:sayHelloToResponse>
    </SOAP-ENV:Body>
</SOAP-ENV:Envelope>
```

That is the same SOAP response as shown in Listing 5 with some HTTP-specific code at the beginning. Since there was no error, the first line produces the code `200`, which in HTTP speak means "everything's OK." If there were any errors/faults while processing the SOAP message (in the header or body), the returned code would generate `500`, which means "internal server error" in HTTP speak. Thus, the first line would look like this:

```
HTTP 500 Internal Server Error
```

The HTTP extension framework

Many applications require services beyond those provided by traditional HTTP. As a result, such applications extend the traditional HTTP protocol. However, these extensions are proprietary to the application itself. The HTTP extension framework attempts to solve that problem by describing a generic extension mechanism for HTTP. Among other things, the HTTP extension framework adds the `M-POST` method, where *M* stands for mandatory. An HTTP request is called a mandatory request if it includes at least one mandatory extension declaration. Include a mandatory extension declaration by using the `Man` or the `C-Man` header fields. The method name of a mandatory request *must* be prefixed by *M-*, hence the mandatory `POST` method is called `M-POST`.

SOAP 1.0 required that a client start off with an HTTP `POST` request and send an `M-POST` request only if the server returned an HTTP error `510`. SOAP 1.1 places no such restriction on a client, thus allowing it to start off with either request type. Below is the same request that we've considered so far presented in `M-POST` form:

Listing 12

```
M-POST http://www.SmartHello.com/HelloApplication HTTP/1.1
Content-Type: text/xml; charset="utf-8"
Content-Length: 587
Man: "http://schemas.xmlsoap.org/soap/envelope/"; ns=01
01-SOAPAction: "http://www.SmartHello.com/HelloApplication#sayHelloTo"
<SOAP-ENV:Envelope
                       xmlns:SOAP-ENV="
http://schemas.xmlsoap.org/soap/envelope/"
                       xmlns:xsi="
http://www.w3.org/1999/XMLSchema-instance"
                       xmlns:xsd="http://www.w3.org/1999/XMLSchema">
     <SOAP-ENV:Header>
     </SOAP-ENV:Header>
    <SOAP-ENV:Body>
         <ns1:sayHelloTo
                      xmlns:ns1="Hello"
                     SOAP-ENV:encodingStyle="
http://schemas.xmlsoap.org/soap/encoding/">
             <name xsi:type="xsd:string">Tarak</name>
         </ns1:sayHelloTo>
    </SOAP-ENV:Body>
</SOAP-ENV:Envelope>
```

As far as the actual SOAP message goes, Listing 12 doesn't differ from Listing 10. The header does feature a few dissimilarities. For example, instead of a `POST` request, we have an `M-POST` request. As described above, every mandatory HTTP request such as `M-POST` requires at least one mandatory extension declaration. Here we have one: the `Man` field describes a mandatory end-to-end extension declaration and maps the header-prefix `01` to the namespace http://schemas.xmlsoap.org/soap/envelope/. Note how this prefix is attached to the `SOAPAction` field.

Once the server processes the request, it will return a response to the client that looks like the code below (assuming that there are no errors):

Listing 13

```
HTTP/1.0 200 OK
Ext:
Content-Type: text/xml; charset="utf-8"
Content-Length: 615
<SOAP-ENV:Envelope
                       xmlns:SOAP-ENV="
http://schemas.xmlsoap.org/soap/envelope/"
                       xmlns:xsi="
http://www.w3.org/1999/XMLSchema-instance"
                       xmlns:xsd="http://www.w3.org/1999/XMLSchema">
     <SOAP-ENV:Body>
          <ns1:sayHelloToResponse
                      xmlns:ns1="Hello"
                      SOAP-ENV:encodingStyle="
http://schemas.xmlsoap.org/soap/encoding/">
                <return xsi:type="xsd:string">Hello John, How are
you doing?</return>
          </ns1:sayHelloToResponse>
    </SOAP-ENV:Body>
</SOAP-ENV:Envelope>
```

Again, the response in Listing 13 resembles the one returned for a `POST` request -- shown in Listing 11 -- except for the `Ext` field.

In using SOAP via HTTP, it is interesting to see that the actual SOAP message (the SOAP envelope and everything within it) always remains the same as the message with no protocol. That fact can be extrapolated to conclude that HTTP is not the only protocol that SOAP works with. For example, SOAP can easily work with the SMTP or any custom homegrown protocol. The only requirement is

that both sides -- the client and the server -- understand the protocol.

SOAP does not define everything

So far I've discussed different aspects that SOAP defines, but there are a number of areas that SOAP does not define. The authors of the SOAP specification explicitly exclude the more involved aspects of building an object model, as well as anything that's already been built. The reason for that can be found upon examining the goals of SOAP. Besides extensibility, a major design goal of SOAP is simplicity. To keep SOAP simple, the authors decided to define only those aspects that were absolutely critical for creating a lightweight protocol. For example, SOAP does not define/specify anything about distributed garbage collection, type safety or versioning, bidirectional HTTP communications, message-box carrying or pipeline processing, object-by-reference, or no-object activation. SOAP is meant to be simple -- a protocol a developer could implement in a couple of days using any programming language on any operating system. When you think about it, this is actually a blessing since SOAP can easily be adapted to existing technologies for building distributed systems, even technologies as different as CORBA and DCOM.

Until next time ...

In this article I introduced you to the basics of SOAP and discussed some of the reasons behind its design. But this is only the tip of the SOAP iceberg. To find out more about SOAP, refer to the Resources section for a link to the SOAP specification. As far as this series goes, what I've included here is all you need to know about the specification.

In Part 2, I will introduce you to Apache's SOAP implementation. With that implementation, you will create a simple distributed application that uses SOAP as the wire protocol.

About the author

A certified Java programmer, Tarak Modi is an architect at North Highland, which specializes in management and technology consulting. He has a master's degree in computer engineering and an MBA concentrating in information systems. He has several years of experience working with C++ and Java and technologies such as DCOM, CORBA, RMI, and EJB. He has written articles for leading software magazines including JavaWorld and is currently working on a book on Java Message Service with Manning Publications. To find out more about Tarak, his articles, and upcoming book, please visit his Website at http://tmodi.home.att.net/

All contents copyright 1995-2010 Java World, Inc. http://www.javaworld.com