Distributed Application Development With SOAP

by Craig Murphy

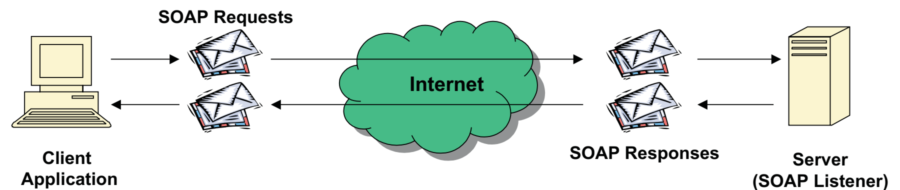
The days of the passive internet are over. Today's internet must be active, responsive, efficient and, very importantly, current. Clients, users, customers and subscribers are not interested in viewing content that is out-of-date or is the same from one day to the next. We can go some way towards a more active internet with technologies such as CGI binaries, ASP pages, Perl and Interactive HTML (iHTML). However, most of these solutions are bound to particular operating systems, thus introducing additional complexity and machine-specific interaction. Ultimately, they can reduce your share of the market.

Would it not be useful if we could work with a language-neutral, operating system-agnostic, mechanism for bringing life to our internet applications? The Simple Object Access Protocol (SOAP) offers us just that: the ability to build distributed applications that add the element of interactivity that we would normally associate with a desktop application.

Over the course of this article I will describe and position SOAP as a mechanism and architecture for building distributed applications that bring together disparate, possibly even legacy, applications. Whilst an understanding of SOAP is enough to start building applications, it is important to realise the benefits of the various toolkits that we can use to build our SOAP applications. This article will demonstrate how we can use the Microsoft SOAP Toolkit 2.0 (Beta 1) to build internet-ready applications using Delphi.

Throughout this article, the term 'Microsoft SOAP Toolkit' relates to the Microsoft SOAP Toolkit 2.0 (Beta 1). The 2.0 Toolkit differs significantly from the 1.0 Toolkit: Microsoft even recommends gradually upgrading applications based on SOAP Toolkit 1.0 to use the new SOAP Toolkit 2.0. Rather than present complete coverage of the SOAP specification and the SOAP Toolkit, this article will concentrate on a high-level overview. I will cover the basics of SOAP and demonstrate how we can use the Microsoft SOAP Toolkit in Delphi.

Normally that would suffice for an article; however, I will also discuss the 'discovery' languages and initiatives, UDDI and WSDL, as both of these acronyms play an important role in any discussion of SOAP and web services. Whilst a complete explanation of WSDL and UDDI is outside the scope of one article, I hope that the explanation I provide will whet your appetite and that you will follow the links in the *Resources* sidebar at the end.



➤ *Figure 1: SOAP Architecture.*

What Is SOAP?

SOAP is a 'structured means of transporting data and method calls between potentially distributed systems'. It is a text-based protocol, thus it does not impose any platform dependencies. Platform independence is promoted further by the use of eXtensible Markup Language (XML) as the messaging format.

SOAP messages that originate from a client application (a Delphi executable or a web page) are known as *SOAP requests*. SOAP requests are received by a server (known as a *SOAP listener*), where they are processed. After the server has processed the request, a *SOAP response* is returned to the client. At each stage of the 'transaction', the SOAP messages are encoded using XML, thus the message can freely pass through firewalls (which has always been a problem for the likes of DCOM and CORBA). Figure 1 graphically represents this sequence of events.

The server-side process that 'actions' a SOAP request is known as a *SOAP endpoint*. The SOAP endpoint ultimately 'knows' how to execute the method specified in the SOAP request. The SOAP endpoint may execute the method by instantiating a DLL on the same machine, or it may pass the SOAP request on to another endpoint (possibly on another server).

Whilst it is not compulsory, SOAP uses HyperText Transfer Protocol (HTTP) as a means of client-server communication. It is possible, however, to use email (SMTP), file transfer protocol (FTP), or even message queuing systems (such as MSMQ) as the transport mechanism. HTTP was chosen because there is an implementation of it on nearly every possible platform. Thus, with a little effort, a small Psion handheld organiser (for example) can create and consume SOAP messages. This was one of the key design issues behind SOAP: 'invent no new technology'. SOAP relies on open specifications and protocols: XML and HTTP are tried and tested and they have no vendor lock-in.

SOAP Message Structure

A SOAP message comprises three XML elements. The first is a compulsory root element called `Envelope`. The second element is a child of the `Envelope` element, it is an optional `Header` element. The third element, also a child of `Envelope`, is a compulsory `Body` element.

The `Envelope` element is required in order to comply with the XML specification: an XML file must have a 'root' element or node that encompasses all child elements or nodes. The `Header` element can be used to carry extra information that is pertinent to the `Body` element. For example, we might include a 'user name' in the `Header` element; it does not affect the meaning of a SOAP message, but it could be used to charge the user based on their usage of a specific method (similar to the Application Service Provider model).

Listing 1 presents a sample SOAP request. The SOAP request contains line numbers; here is an explanation of each line:

1. The opening `<SOAP-ENV:Envelope>` element. It will hold the rest of the SOAP message. The prefix `SOAP-ENV` is an XML namespace. Namespaces are required to ensure that clashes between element names don't occur.

2. This is the opening `<SOAP-ENV:Body>` element. Again, it is part of the SOAP message, so it must be prefixed with the `SOAP-ENV` namespace.

3. This is the opening element for `<m:getCustAddress>`. Inside the `Body` element is the message content. This can be anything you like, as long as it is valid XML. In this case I have defined a new namespace called `m`. The namespace itself is a Universal Resource Identifier (a URI). The URI doesn't have to be resolvable either: it can be unreachable!

4. This is the `<CustNo>` element; it contains the customer number 1221.

5. This is the closing element for `</m:getCustAddress>`.

6. This is the closing element for `</SOAP-ENV:Body>`.

7. This is the closing element for `</SOAP-ENV:Envelope>`.

```
1:<SOAP-ENV:Envelope xmlns:SOAP-ENV="http://schemas.xmlsoap.org/soap/envelope">
2:<SOAP-ENV:Body>
3:<m:getCustAddress xmlns:m="http://www.itecuk.com/dbdemos/customer">
4:<CustNo>1221</CustNo>
5:</m:getCustAddress>
6:</SOAP-ENV:Body>
7:</SOAP-ENV:Envelope>
```

➤ *Above: Listing 1*

➤ *Below: Listing 2*

```
1:<SOAP-ENV:Envelope xmlns:SOAP-ENV="http://schemas.xmlsoap.org/soap/envelope">
2:<SOAP-ENV:Body>
3:<m:getCustAddressResponse xmlns:m="http://www.itecuk.com/dbdemos/customer">
4:<CustNo>1221</custno>
5:<Company>Kauai Dive Shoppe</Company>
6:<Addr1>4-976 Sugarloaf Hwy</Addr1>
7:<Addr2>Suite 103</Addr2>
8:<City>Kapee Kauai</City>
9:<State>HI</State>
10:<Zip>94766-1234</Zip>
11:<Country>US</Country>
12:</m:getCustAddressResponse>
13:</SOAP-ENV:Body>
14:</SOAP-ENV:Envelope>
```

If the SOAP request (Listing 1) was sent to a server (one that implements the `getCustAddress` method), then the SOAP response might resemble Listing 2. Again, I have added line numbers to the SOAP response. Let's go through each one:

1. This is exactly the same as the SOAP request.

2. This is exactly the same as the SOAP request.

3. This is the same as the SOAP request, except this time, because we're dealing with a response, the word `Response` is tagged onto the element name.

4. Lines 4 to 11 form part of the returned message.

12. The closing element for `<m:getCustAddressResponse>`.

13. This is exactly the same as the SOAP request.

14. This is exactly the same as the SOAP request.

Web Services

Attend any technical IT conference this year (or last year for that matter) and you will hear the phrase 'web services'. The internet application community are using a 'web service' to mean a server-side method that can be executed from a possibly remote client application using little more than a web browser. This is exactly what SOAP is about: the execution or invocation of web services. I use the word 'possibly' to mean that you may wish to implement SOAP-based applications on your corporate intranet; it is a great way of bringing together data from more than one system, or for the amalgamation of data from legacy databases.

However, how do we know that a particular URL offers web services? There is a lot of work being carried out around the notion of 'discovery'; that is, the discovery of web services. In essence, we need a mechanism that allows client applications the opportunity to find out what web services are available from a given website. The Web Service Description Language (WSDL) goes some way towards a solution for the problem of discovery. WSDL is yet another internet acronym that we have to contend with.

What Is WSDL?

WSDL, like SOAP, is encoded using XML, therefore it too can be considered platform-neutral. WSDL provides a means of describing the web services a particular 'object' has to offer. I use the word 'object' loosely: how the web services are implemented is up to the server-side developer. As client-side 'users' or consumers of the service, all we are interested in is the interface to the web service.

WSDL is almost analogous to Interface Definition Language (IDL). Like IDL, WSDL allows us to specify data types. If you scan the WSDL in Listing 4 you'll see references to 'double', and you'll notice the reference to `CalcITEC.xsd`.

```
unit ITECCalc;
interface
uses
  ComObj, ActiveX, ITEC_TLB, StdVcl;
type
  TCalc = class(TAutoObject, ICalc)
  protected
    function Add(A,B: Double): Double; safecall;
    function Divide(A, B: Double): Double; safecall;
    function Multiply(A, B: Double): Double; safecall;
    function Subtract(A, B: Double): Double; safecall;
    { Protected declarations }
  end;
implementation
uses ComServ;
function TCalc.Add(A,B: Double): Double;
begin
  Add := A + B;
end;
function TCalc.Divide(A, B: Double): Double;
begin
  if (A <> 0) AND (B <> 0) then
    Divide := A / B
  else
    Divide := 0;
end;
function TCalc.Multiply(A, B: Double): Double;
begin
  Multiply := A * B;
end;
function TCalc.Subtract(A, B: Double): Double;
begin
  Subtract:= A - B;
end;
initialization
  TAutoObjectFactory.Create(ComServer, TCalc, Class_Calc,
    ciMultiInstance, tmApartment);
end.
```

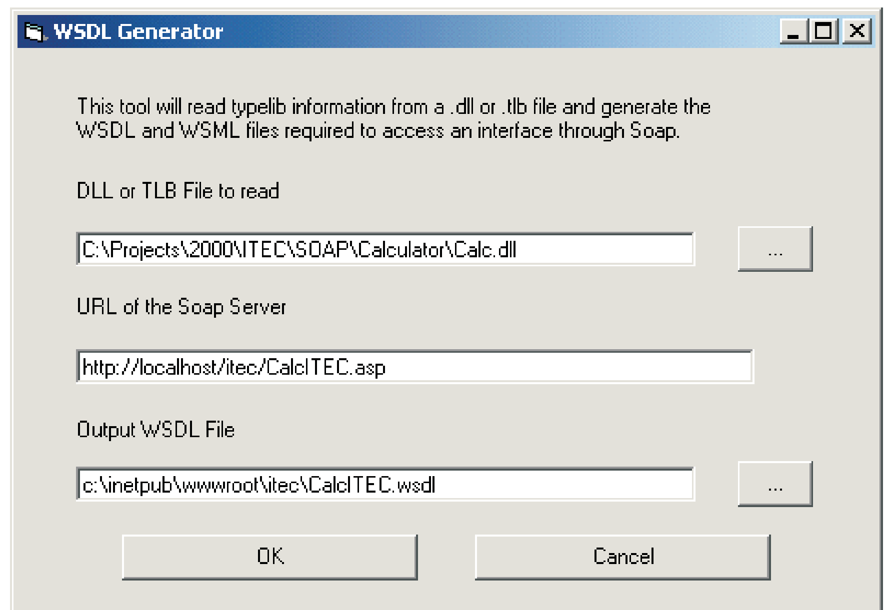
➤ *Listing 3*

WSDL uses XML Schema (Part 2) to specify data types. Unfortunately, an explanation of XML Schema is outside the scope of this article; however, if you are interested, there is a URL in the *Resources* sidebar. In a nutshell, WSDL provides us (or a SOAP toolkit) with enough information to construct a SOAP request and decipher the SOAP response.

Typically, a client application requests a WSDL document that describes the web services on offer. Already we can see it is not a perfect solution, as the client still has to have some knowledge: it must know the URL and the name of the WSDL file that describes the web services on offer. It is not uncommon for one web server to have more than one WSDL document!

The Microsoft SOAP Toolkit can create WSDL from DLLs and/or type libraries. Thus, as developers, we do not have to create WSDL documents by hand: that would be a daunting task!

However, Microsoft has introduced yet another layer of abstraction into the web services equation: WSML, Web Service Meta Language. Like WSDL, WSML is encoded using XML. WSML is used in conjunction with WSDL and provides a mechanism for describing the server-side implementation of the web services. As you might imagine, a WSML document identifies the `PROGID` of the DLL that provides the web services. The same process that creates WSDL from a DLL also creates WSML.



➤ *Figure 2: wsdlgen.exe in action.*

I must stress that WSML is unique to the Microsoft SOAP Toolkit and, given that it is a server-side extension, it does not change the client-side operation at all. Essentially, the WSML document provides a means of mapping the high-level web service on to a concrete implementation of the method required to satisfy the request.

Building Web Services With Delphi (Server-Side)

We can use Delphi to build server-side web services: Listing 3 presents the code required to implement a simple calculator. The iTec calculator offers four web services: `Add`, `Subtract`, `Divide` (with divide by zero protection) and `Multiply`. Each method takes two parameters of type `double` and returns an answer, also of type `double`. The iTec calculator is implemented as a DLL: you will need to create an ActiveX Library and then an OLE Automation object to build your own DLLs. You will also need to register the DLL: this can be achieved using the `Register ActiveX Server` option on to the `Run` menu.

Microsoft has provided a standalone tool that creates WSDL and WSML from a DLL: it is called wsdlgen.exe. As long as the DLL is registered, this executable allows us to build WSDL and WSML based on the information we provide:

- ➢ The location of a DLL that implements the web services (calc.dll).
- ➢ The SOAP listener URL (this will be used in the WSDL file).
- ➢ The directory where the WSDL and WSML should be created.

Figure 2 depicts the WSDL generator in action: I use localhost because I am running both the client and the server on the same

machine. If you are fortunate enough to have a development server available, then you may change localhost to suit your environment. Assuming that we have run the iTec calculator through wsdlgen.exe, Listing 4 presents the WSDL required for the iTec calculator object; Listing 5 presents the WSML.

The Microsoft SOAP Toolkit provides us with server-side objects that will help us process our SOAP requests and responses. The `MSSOAP.SoapServer` object allows us to pass it WSDL and WSML documents that describe the methods a SOAP request might contain. Thus, because we are server-side, we know that the traditional ASP `Request` and `Response` objects are available for us to use. SOAP over HTTP uses the ASP `Request` object to carry the SOAP request (as XML), and therefore the SOAP Toolkit allows us to pass the ASP `Request` object as a parameter to the `Invoke` method of the `SoapServer` object. Similarly, the ASP `Response` object can also be passed in to the `Invoke` method. Thus, we can assume that the `Invoke` method performs the following:

- ➢ Decodes the SOAP request (via the ASP `Request` object) and identifies the method to be invoked.
- ➢ Creates an instance of the object that implements the method (from the WSML and the DLL).
- ➢ Invokes the method using the parameters supplied in the SOAP request.
- ➢ Constructs the correct SOAP response and creates the correct ASP `Response` object in return.

I have chosen to implement the 'listener' using an Active Server Page: the language is VBScript. However,

➤ *Listing 4*

```
<definitions  name ='CalcITEC'   targetNamespace =
  'http://localhost/itec/CalcITEC.wsdl'
    xmlns:tns='http://localhost/itec/CalcITEC.wsdl'
    xmlns:xsd1='http://localhost/itec/CalcITEC.xsd'
    xmlns:soap='http://schemas.xmlsoap.org/wsdl/soap/'
    xmlns='http://schemas.xmlsoap.org/wsdl/'>
  <types>
    <schema targetNamespace=
      'http://localhost/itec/CalcITEC.xsd'
      xmlns='http://www.w3.org/1999/XMLSchema'>
    </schema>
  </types>
  <message name='Add'>
    <part name='A' type='double'/>
    <part name='B' type='double'/>
  </message>
  <message name='AddResponse'>
    <part name='Result' type='double'/>
  </message>
  <message name='Subtract'>
    <part name='A' type='double'/>
    <part name='B' type='double'/>
  </message>
  <message name='SubtractResponse'>
    <part name='Result' type='double'/>
  </message>
  <message name='Multiply'>
    <part name='A' type='double'/>
    <part name='B' type='double'/>
  </message>
  <message name='MultiplyResponse'>
    <part name='Result' type='double'/>
  </message>
  <message name='Divide'>
    <part name='A' type='double'/>
    <part name='B' type='double'/>
  </message>
  <message name='DivideResponse'>
    <part name='Result' type='double'/>
  </message>
  <portType name='CalcITECPortType'>
    <operation name='Add' parameterOrder='AddInput1
      AddInput2'>
      <input message='tns:Add' />
      <output message='tns:AddResponse' />
    </operation>
    <operation name='Subtract'
      parameterOrder='SubtractInput1 SubtractInput2'>
      <input message='tns:Subtract' />
      <output message='tns:SubtractResponse' />
    </operation>
    <operation name='Multiply'
      parameterOrder='MultiplyInput1 MultiplyInput2'>
      <input message='tns:Multiply' />
      <output message='tns:MultiplyResponse' />
    </operation>
    <operation name='Divide' parameterOrder='DivideInput1
      DivideInput2'>
      <input message='tns:Divide' />
      <output message='tns:DivideResponse' />
    </operation>
  </portType>
  <binding name='CalcITECBinding'
    type='tns:CalcITECPortType' >
    <soap:binding style='document'
      transport='http://schemas.xmlsoap.org/soap/http' />
    <operation name='Add' >
      <soap:operation
        soapAction='http://localhost/itec/CalcITEC.asp' />
      <input>
        <soap:body use='encoded'
          namespace='http://localhost/itec/CalcITEC.xsd'
          encodingStyle='http://schemas.xmlsoap.org/soap/
          encoding/' />
      </input>
      <output>
        <soap:body use='encoded'
          namespace='http://localhost/itec/CalcITEC.xsd'
          encodingStyle='http://schemas.xmlsoap.org/soap/
          encoding/' />
      </output>
    </operation>
    <operation name='Subtract' >
      <soap:operation soapAction=
        'http://localhost/itec/CalcITEC.asp' />
      <input>
        <soap:body use='encoded' namespace=
          'http://localhost/itec/CalcITEC.xsd'
          encodingStyle='http://schemas.xmlsoap.org/soap/
          encoding/' />
      </input>
      <output>
        <soap:body use='encoded'
          namespace='http://localhost/itec/CalcITEC.xsd'
          encodingStyle='http://schemas.xmlsoap.org/soap/
          encoding/' />
      </output>
    </operation>
    <operation name='Multiply' >
      <soap:operation
        soapAction='http://localhost/itec/CalcITEC.asp' />
      <input>
        <soap:body use='encoded'
          namespace='http://localhost/itec/CalcITEC.xsd'
          encodingStyle='http://schemas.xmlsoap.org/soap/
          encoding/' />
      </input>
      <output>
        <soap:body use='encoded'
          namespace='http://localhost/itec/CalcITEC.xsd'
          encodingStyle='http://schemas.xmlsoap.org/soap/
          encoding/' />
      </output>
    </operation>
    <operation name='Divide' >
      <soap:operation
        soapAction='http://localhost/itec/CalcITEC.asp' />
      <input>
        <soap:body use='encoded'
          namespace='http://localhost/itec/CalcITEC.xsd'
          encodingStyle='http://schemas.xmlsoap.org/soap/
          encoding/' />
      </input>
      <output>
        <soap:body use='encoded'
          namespace='http://localhost/itec/CalcITEC.xsd'
          encodingStyle='http://schemas.xmlsoap.org/soap/
          encoding/' />
      </output>
    </operation>
  </binding>
  <service name='CalcITEC' >
    <port name='CalcITECPortType'
      binding='tns:CalcITECBinding' >
      <soap:address
        location='http://localhost/itec/CalcITEC.asp' />
    </port>
  </service>
</definitions>
```

```
<servicemapping name='CalcITEC'>
  <service name='CalcITEC'>
    <using PROGID='ITEC.Calc' cachable='0'
      ID='CalcITECObject' />
   <port name='CalcITECPortType'>
    <operation name='Add'>
      <execute uses='CalcITECObject' method='Add'
        dispID='1'>
        <parameter callIndex='1' name='A' elementName='A' />
        <parameter callIndex='2' name='B' elementName='B' />
        <parameter callIndex='-1' name='retval'
          elementName='Result' />
      </execute>
    </operation>
    <operation name='Subtract'>
      <execute uses='CalcITECObject' method='Subtract'
        dispID='6'>
        <parameter callIndex='1' name='A' elementName='A' />
        <parameter callIndex='2' name='B' elementName='B' />
        <parameter callIndex='-1' name='retval'
          elementName='Result' />
      </execute>
    </operation>
    <operation name='Multiply'>
      <execute uses='CalcITECObject' method='Multiply'
        dispID='5'>
        <parameter callIndex='1' name='A' elementName='A' />
        <parameter callIndex='2' name='B' elementName='B' />
        <parameter callIndex='-1' name='retval'
          elementName='Result' />
      </execute>
    </operation>
    <operation name='Divide'>
      <execute uses='CalcITECObject' method='Divide'
        dispID='4'>
        <parameter callIndex='1' name='A' elementName='A' />
        <parameter callIndex='2' name='B' elementName='B' />
        <parameter callIndex='-1' name='retval'
          elementName='Result' />
      </execute>
    </operation>
  </port>
    </service>
  </servicemapping>
```

➤ *Listing 5*

you can see that the same VBScript would suffice for any generic SOAP listener, so as Delphi developers we can use these few lines of VBScript to provide a conduit to our DLL-based web services. Listing 6 presents the few lines of VBScript required to invoke the SOAP Server object.

Using Web Services In Delphi (Client-Side)

Testing our web services requires us to build a client-side application. Figure 3 presents the main form of an application that allows us to test our web services.

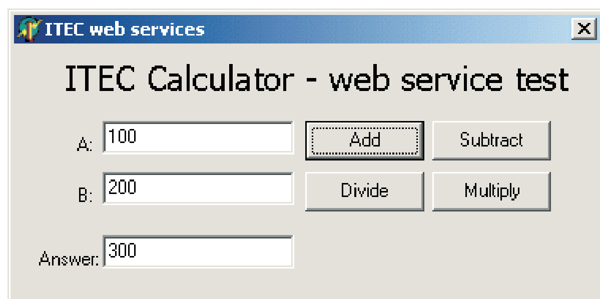
The Microsoft SOAP Toolkit provides us with client-side support in the form of the `MSSOAP.SOAPClient` object. We must instantiate the `SOAPClient` object using late binding; after all we are going to be dynamically calling methods which are as yet unknown at runtime.

The `SOAPClient` object allows us to load a WSDL document (one that describes the calculator web services) using the `mssoapinit` method:

```
ovSOAPClient.mssoapinit(
  'http://localhost/itec/
  CalcITEC.wsdl', 'CalcITEC',
  'CalcITECPortType');
```

Once `CalcITEC.wsdl` is loaded, the `SOAPClient` locates the `CalcITEC` service element, as shown in Listing 7. The service element provides information relating to the SOAP endpoint: `CalcITEC.asp`; this means the `SOAPClient` knows where to send SOAP requests. `CalcITEC.asp` is the server-side process that is able to service SOAP requests for the iTec calculator.

The last parameter, `CalcITECPortType`, is used to obtain the specification for each possible method. Listing 8 provides the specification for the `Add` method. The specification for a method includes information about what a SOAP request needs to provide in



➤ *Figure 3: Web service test application; it all adds up.*

```
<%
  Dim SoapServer
  Dim WSDLFilePath
  Dim WSMLFilePath

  On Error Resume Next
  Response.ContentType = "text/xml"

  Set SoapServer = Server.CreateObject("MSSOAP.SoapServer")

  WSDLFilePath = Server.MapPath("CalcITEC.wsdl")
  WSMLFilePath = Server.MapPath("CalcITEC.wsml")
  SoapServer.Init WSDLFilePath, WSMLFilePath

  SoapServer.SoapInvoke Request, Response
%>
```

➤ *Above: Listing 6* ➤ *Below: Listing 7*

```
<service name='CalcITEC' >
  <port name='CalcITECPortType' binding='tns:CalcITECBinding' >
    <soap:address location='http://localhost/itec/CalcITEC.asp' />
  </port>
</service>
```

```
<portType name='CalcITECPortType'>
  <operation name='Add' parameterOrder='AddInput1 AddInput2'>
    <input message='tns:Add' />
    <output message='tns:AddResponse' />
  </operation>
</portType>
```

➤ *Listing 8*

```
<message name='Add'>
  <part name='A' type='double'/>
  <part name='B' type='double'/>
</message>
```

➤ *Listing 9*

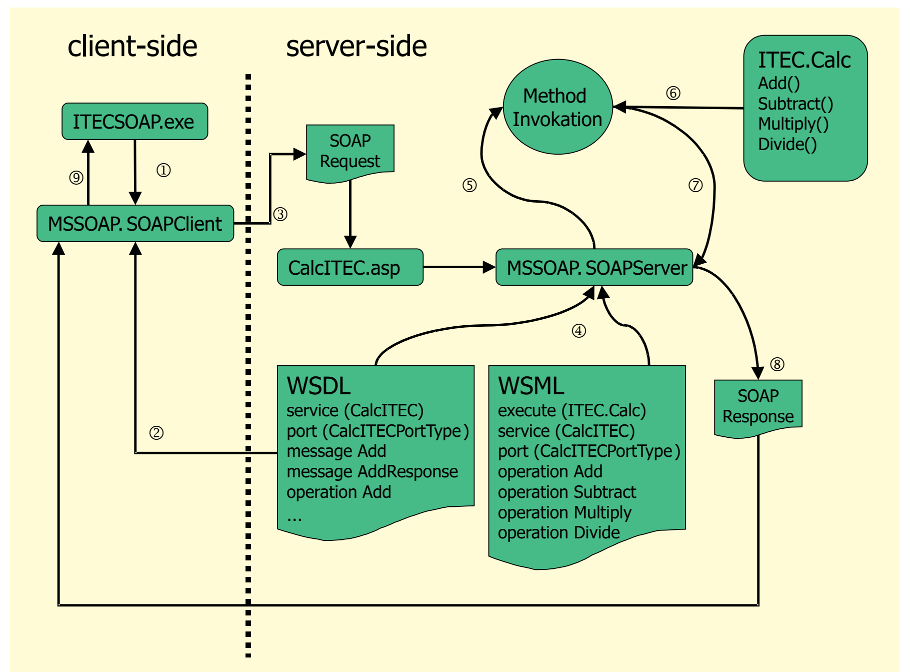
```
<message name='AddResponse'>
  <part name='Result'
type='double'/>
</message>
```

➤ *Listing 10*

order to execute the method. Listing 9 presents the information required to build a SOAP request for the `Add` method. Once a SOAP request for the `Add` method has been executed, the `SOAPClient` can expect a SOAP response formatted using the information provided by Listing 10.

The complete end-to-end process is best described graphically: Figure 4 depicts the client and server-side activity required to process a SOAP request.

1. The client application instantiates `MSSOAP.SOAPClient`.
2. The `SOAPClient` is initialised with the WSDL for the iTec calculator.
3. Client-side method invocation causes the `SOAPClient` to generate a SOAP request that is sent to `CalcITEC.asp`.
4. CalcITEC.asp is listening for SOAP requests: when one is received, it instantiates `MSSOAP.SOAPServer` and loads the iTec calculator WSDL and WSML.
5. `CalcITEC.asp` then asks the `SOAPServer` to execute the method specified in the ASP `Request` object.
6. Under the hood, `SOAPServer` instantiates `ITEC.Calc` and executes the required method.
7. The response from `ITEC.Calc` is returned to the SOAPServer.



➤ *Figure 4: Client and server-side activity to process a SOAP request.*

8. A SOAP response is created and returned to the `SOAPClient`.
9. The `SOAPClient` 'deciphers' the response and passes it to the client.

Whilst this may seem like an awful lot of work, we have actually done very little. In fact, we have written very little SOAP-specific code ourselves; most of our development time has been spent building a DLL to solve a problem, and building a test application for that DLL. This is the beauty of SOAP: good toolkits and libraries can shield us from the complexities of XML parsing, HTTP and SOAP message construction.

As an interesting aside, the `SOAPServer` object actually uses XPath expressions to address the WSDL and WSML documents (I introduced XPath expressions in *The Delphi Magazine*, Issue 66: the article is called *Using XML And XSLT In Delphi*).

Listing 11 provides the full source for the iTec calculator. We must use the `ComObj` unit if late binding is to be achieved, `CreateOleObject` is implemented in that unit. So, the late bound methods (those bound at runtime), really should have good error checking. For example, what if the `Add` method doesn't exist? What if I call the `Add` method with two strings? SOAP provides a means of 'error' handling: *SOAP faults*. A SOAP fault is simply a SOAP message (encoded using XML) that includes information about any errors that may have occurred. Syntax errors, like type mismatch, are reported using SOAP faults.

Why Use A SOAP Toolkit?

Implementing a SOAP-based application using raw XML packets

Installing The iTec Calculator

I have used Windows 2000 Server and Internet Information Services (IIS) to build and test the iTec calculator example. The Microsoft SOAP Toolkit 2.0 beta 1 client-side objects run on Microsoft Windows 98, Microsoft Windows ME, Microsoft Windows NT 4.0 Service Pack 6, and Microsoft Windows 2000 Service Pack 1. The server-side objects run on Active Server Pages (ASP) pages on Windows 2000 and Windows NT 4.0 Service Pack 6. Unfortunately Internet Explorer 5.0 or higher is required: MSXML 3.0 forms part of the SOAP Toolkit 2.0 installation process.

The example code that accompanies this article requires careful installation: both client and server must have the SOAP Toolkit 2.0 installed.

Server-side:

1. You will need to create a virtual directory called itec in your IIS wwwroot directory.
2. Copy ITECCalc.wsdl, ITECCalc.wsml and ITECCalc.asp into the itec directory.
3. Compile and register Calc.dll.

The client-side installation is much easier: simply compile and run ITECSOAP.dpr.

between client application and server listeners is perfectly possible; however, the creation and deciphering of SOAP requests and responses is an overhead. Both client and server need to know how to work with HTTP, XML and SOAP, and that is before the application gets to see any of the method calls and data. Thus the toolkit provides the abstraction we need: as developers we are able to concentrate on the application in hand, and we do not have to worry about the creation of SOAP requests and the deciphering of SOAP responses.

A Few Words About UDDI

Whilst WSDL works, clients still have to interrogate server-side WSDL files in order to determine the web services a website may offer. In our example, the client application knew http://localhost/itec/CalcITEC.wsdl existed, and it knew what methods to use. With more and more web services being developed, it is clear than we will need some means of indexing the services, and a mechanism that allows 'blind' clients to 'discover' the services offered by a particular website. Universal, Description, Discovery and Integration (UDDI) takes things to the next stage. UDDI is an internet-based registry of web services that uses SOAP for the registration of web services, and SOAP for the subsequent discovery (or lookup). Like SOAP, UDDI is a collaborative project involving Microsoft, IBM and Ariba.

UDDI revolves around the idea of a collection of Business Registries, organised just like a telephone book: White Pages provide a list of web services, Yellow Pages are organised by business sector, Green Pages include 'programmatic' descriptions, thus discovery of services can be automated.

UDDI also entails a Service Type Registration which provides a detailed description of a web service.

Summary

Over the course of this article I have tried to provide an insight into the world of web services. As you have probably gathered, there is a lot to learn. SOAP, XML and HTTP for starters! By using SOAP toolkits, we can alleviate some of the burden we face: the toolkits provide a layer of abstraction to the underlying protocols. The abstractions provided can also protect us from change. SOAP, XML and HTTP are versioned specifications (SOAP is at version 1.1). Should the underlying specification change, the toolkits should absorb many of the changes required, thus changes to our own client-side code should be minimal.

The toolkits also allow us, as developers, to concentrate on solving the problem in hand. If we had to develop the code that would be required to construct SOAP messages, send SOAP messages, and invoke the methods, we would find ourselves spending a lot of time writing code that is peripheral to the problem we are trying to solve. We do have to endure a slight shift in our approach: rather than write lengthy multi-purpose functions, we should try and make our functions as granular as possible. That is, make each routine perform just one specific task. In doing so, we are hopefully opening the door to global code re-use. Given that one client-side SOAP request can spawn more than one server-side SOAP request, the notion of using

➤ *Listing 11*

```
unit CalcTest;
interface
uses
  Windows, Messages, SysUtils, Classes, Graphics,
  Controls, Forms, Dialogs, StdCtrls;
type
  TForm1 = class(TForm)
    btnAdd: TButton;
    btnDivide: TButton;
    btnMultiply: TButton;
    btnSubtract: TButton;
    edtA: TEdit;
    edtB: TEdit;
    edtAnswer: TEdit;
    Label1: TLabel;
    Label2: TLabel;
    Label3: TLabel;
    Label4: TLabel;
    procedure btnAddClick(Sender: TObject);
    procedure btnDivideClick(Sender: TObject);
    procedure btnSubtractClick(Sender: TObject);
    procedure btnMultiplyClick(Sender: TObject);
  private
  public
  end;
var Form1: TForm1;
implementation
{$R *.DFM}
uses
  ComObj;
procedure TForm1.btnAddClick(Sender: TObject);
var ovSOAPClient : Olevariant;
  a,b, answer : double;
begin
  ovSOAPClient := CreateOleObject('MSSOAP.SoapClient');
  ovSOAPClient.mssoapinit(
    'http://localhost/itec/CalcITEC.wsdl', 'CalcITEC',
    'CalcITECPortType');
  a :=  StrToFloat(edtA.Text);
  b       :=  StrToFloat(edtB.Text);
  answer  :=  ovSOAPClient.Add (a,b);
  edtAnswer.Text := FloatToStr(answer);
end;
procedure TForm1.btnDivideClick(Sender: TObject);
var ovSOAPClient : Olevariant;
  a,b, answer : double;
begin
  ovSOAPClient := CreateOleObject('MSSOAP.SoapClient');
  ovSOAPClient.mssoapinit(
    'http://localhost/itec/CalcITEC.wsdl', 'CalcITEC',
    'CalcITECPortType');
  a :=  StrToFloat(edtA.Text);
  b :=  StrToFloat(edtB.Text);
  answer   :=  ovSOAPClient.Divide (a,b);
  edtAnswer.Text := FloatToStr(answer);
end;
procedure TForm1.btnSubtractClick(Sender: TObject);
var ovSOAPClient : Olevariant;
  a,b, answer : double;
begin
  ovSOAPClient := CreateOleObject('MSSOAP.SoapClient');
  ovSOAPClient.mssoapinit(
   'http://localhost/itec/CalcITEC.wsdl', 'CalcITEC',
   'CalcITECPortType');
  a       :=  StrToFloat(edtA.Text);
  b       :=  StrToFloat(edtB.Text);
  answer  :=  ovSOAPClient.Subtract (a,b);
  edtAnswer.Text := FloatToStr(answer);
end;
procedure TForm1.btnMultiplyClick(Sender: TObject);
var ovSOAPClient : Olevariant;
  a,b, answer : double;
begin
  ovSOAPClient := CreateOleObject('MSSOAP.SoapClient');
  ovSOAPClient.mssoapinit(
    'http://localhost/itec/CalcITEC.wsdl',
    'CalcITEC', 'CalcITECPortType');
  a       :=  StrToFloat(edtA.Text);
  b       :=  StrToFloat(edtB.Text);
  answer  :=  ovSOAPClient.Multiply (a,b);
  edtAnswer.Text := FloatToStr(answer);
end;
end.
```

SOAP for aggregation promotes the granular theory even more.

An understanding of SOAP puts you in good stead for the not too distant future. The recent release of Microsoft .NET takes SOAP a stage further: SOAP is omnipresent in .NET, therefore choosing to ignore it may prove costly. Similarly, Microsoft uses SOAP as the underlying message transport protocol in its BizTalk Server 2000 product. BizTalk Server processes BizTalk documents: these are XML documents that adhere to the BizTalk Document and Messaging Specification (the BizTalk Framework). BizTalk is a key player in the B2B and e-commerce arena: it provides a mechanism that allows the exchange of documents between trading partners. Whilst BizTalk itself is a Microsoft product, the BizTalk Framework is an open specification, therefore you are able to avoid vendor lock-in.

Finally, if you are using Internet Explorer 5 or higher and wish to see a simpler SOAP example, point your browser here:

```
www.craigmurphy.com/SOAP/
  SimpleSOAPClient.htm
```

Resources

Useful URLs

The Microosft SOAP Toolkit 2.0 beta 2 is now available at http://msdn.microsoft.com/code/sample.asp?url=/msdn-files/027/001/580/msdncompositedoc.xml

The previous SOAP Toolkit is at http://msdn.microsoft.com/downloads/default.asp?URL=/code/sample.asp?url=/MSDN-FILES/027/001/529/msdncompositedoc.xml

Promoters of SOAP: www.lucin.com, www.iona.com

UDDI: www.uddi.org

The SOAP Specification: http://msdn.microsoft.com/xml/general/soaptemplate.asp

The Microsoft XML Developer Center: http://msdn.microsoft.com/ xml/

The .NET framework: http://msdn.microsoft.com/net/

The BizTalk website: www.biztalk.org

XML Schema Part 2: Data Types: www.w3.org/TR/xmlschema-2/

Useful Reading

XML and SOAP Programming with BizTalk Servers, Brian E. Travis, Microsoft Press, ISBN 0-7356-1126-2.

Understanding SOAP, Kennard Scribber & Mark C.Stiver, Sams, ISBN 0-672-31922-5.

Craig Murphy works as an Enterprise Developer for Currie & Brown (www.currieb.com); their primary business is quantity surveying, cost management and project management. Email Craig at Craig.Murphy@currieb.co.uk or Craig@isleofjura.demon.co.uk