Connecting Web Services: C++/ Systinet (Idoox) to ASP/VBScript

Zoran Zaev

DOWNLOAD

This month, Zoran Zaev builds a Web Service in C++ using Systinet's WASP for C++ (formerly known as IdooXoap for C++), and then calls it from ASP/VBScript. This article demonstrates the kinds of issues that you can run into as you work on implementing Web Services in C++, using WSDL with C++, and calling this implementation from an environment like ASP.

IN my previous few articles, I showed you how to connect different implementations of Web Services, from Java and IBM to Microsoft and Visual Basic, and even from scripting environments like Perl, VBScript, and JScript. This month, I'll re-create my sample Web Service from the past articles (a job bank service, where job postings can be added by your company's business partners company); however, now I'll use C++ for the server side. I'll call this service both from a C++ client and from an ASP (Active Server Pages) client using VBScript.

WASP for C++ package setup

I decided to use Systinet's (formerly known as Idoox) SOAP implementation for my C++ service. The package is called WASP for C++, and you can find Systinet's site at www.systinet.com/index.html. There are a few other C++ implementations on the market that you could use. I picked Systinet's because it's one of the better-known SOAP implementations. Systinet provides both Java and C++ Web Services implementations, as well as Web Services extensions for Forte. They also have a UDDI implementation that's implemented as an extension to the Java Web Services implementation.

Systinet's C++ SOAP implementation comes in two flavors, Lite and Advanced. The Lite version is an open source C++ implementation for Windows 2000, Windows CE, Linux (Debian and RedHat), and Sun Solaris operating systems. The sources are also available from Systinet, so WASP for C++ can be ported to other platforms. WASP for C++ Lite is free for commercial use, unlike the Advanced implementation, which is an enterprise-level platform. The Advanced implementation is free only for development and testing and provides some additional features over the Lite implementation (for example, support for SSL security).

You can download either version from www.systinet.com/download.html. You're required to register with the site first (it's free).

The current version of WASP for C++ uses a compiler written in Java to produce stubs and skeletons from service description files (Java is also used for the graphical installation wizard). So, if you're going to perform these tasks you need to install the JDK 1.3.x or later (available from http://java.sun.com/j2se/1.3/). See my article in the December 2001 issue of *XML Developer* for more on Java Web Services, JDK installation, and the like.

The installation of WASP for C++ is fairly easy and quick, particularly if you're using the graphical installation wizard. There will be some differences from operating system to operating system, of course. In my case, I installed it on a Windows 2000 computer just by double-clicking on the downloaded file.

I installed the package in the default folder of C:\Program Files\wasp_cpp. Once installed, you can access the package's documentation at C:\Program Files\wasp_cpp\doc\products\waspc\index.html (if you selected the default installation location under Windows). For more information about the installation of WASP Lite on other operating systems, see www.systinet.com/products/wasp_lite/doc/index.html, or the documentation that came within the downloaded file (for example, Linux and Solaris would need two environment variables, WASPC_HOME and LD_LIBRARY_PATH, to be set, so make sure you check the installation instructions for your operating system).

WASP for C++ 1.2 supports SOAP 1.1 and WSDL 1.0. However, some exceptions apply, such as:

- SOAP headers aren't processed and are ignored.
- All XML Schema datatypes are handled, but some are treated as strings (for example, dates). Your program will have to handle the conversion.
- WASP for C++ generates the "xsi:type" attribute, and when it receives the type tag, it processes them. As a result, your xsi:type attributes have to match to what you have in the WSDL file.

For more information on the limitations of the current release, see the latest release notes. Also, keep in mind that if you create your WSDL file using utilities that

support WSDL 1.1, you may have to adjust it. The WASP for C++ server only supports the WSDL 1.0 specification (I'll show you some of the differences later in this article).

To compile the files on Windows 2000, I used Visual Studio 6 SP4 and its C++ compiler. The document download contains the Visual Studio projects and most of their settings. However, don't forget to check the following for proper compiling of your files (especially the first two settings, which may not be present in your Visual Studio environment):

- Under the Tools | Options menu, on the Directories tab, add an additional include file to the wasp_cpp\libs\include directory. My entry is C:\PROGRAM FILES\WASP_CPP\LIBS\INCLUDE.
- Within the same location in Visual Studio, add a library include setting to the wasp_cpp\libs\bin\wasp directory. My entry is C:\PROGRAM FILES\WASP_CPP\LIBS\BIN\WASP.

The server, client, and stubs projects should have the following settings (if you use the projects within the code download, this should be preset for you):

- Under Project Settings, on the Link tab, under the General categories pull-down menu, you should have "WASPDbg.lib" for Object/ Library Modules; under the Input pull-down menu, you should set to ignore the libraries nafxcwd.lib,MSVCRT; and an additional library path should be set to something like C:\Program Files\wasp_cpp\libs\bin\wasp.
- Under Project Settings, on the C++ tab, under the Preprocessor categories pull-down menu, for additional include directories you should have something like C:\Program Files\wasp_cpp\libs\include.

I created four projects:

- The stub files with clsJobsSoapPortImpl, JobBank, JobBankIA, and JobBankStructs cpp and header files
- The server files containing the clsJobsSoapPortImpl and server files with dependence on the stubs project
- A MakeFile project with dependencies on both the server and the client project
- The client.cpp file and a dependence on the stubs project

With this setup, you should be able to compile your files. For more information on building your WASP for C++ projects, especially for Linux and Solaris, see the instructions provided with the HelloWorld example at www.systinet.com/products/waspc/tutorial/helloworld.html.

The installation for the WASP for C++ package, as described in the previous section, is the same for both a client-side and a server-side Web Services setup. However, there are some unique steps to building the client-side and server-side Web Services implementations.

Web Service server-side implementation

The first thing to do in the creation of the WASP for C++ server-side Web Services is to create your own WSDL file that describes your service. If you're creating a client-side implementation only, then you'll likely have a WSDL file provided to you by the person or organization that's hosting the Web Service.

There are a few options when it comes to obtaining a WSDL file:

- You can write your own WSDL file by hand, if you're familiar with the basic WSDL syntax (see the W3C specification at www.w3.org/TR/wsdl).
- Systinet suggests using its Java2WSDL generator utility that's included with WASP for Java, to create a WSDL file from a simple Java interface or class that mimics your service interfaces. You can obtain the WASP for Java from www.systinet.com/products/wasp_lite/index.html.

For this article, I used the WSDL file that I created using the Microsoft WSDL generator utility. There were a few modifications that I had to make by hand:

- Make sure that the soapAction attribute of the <soap:operation> element is empty:

```
<soap:operation soapAction='' />
```

  Microsoft typically provides entries within this attribute.
- The location attribute within the <soap:address> element should be adjusted to match your server-side system configuration. For example, the value I used was:

```
<soap:address location=
'http://localhost:7777/wasp/jobadd' />
```

- In the location attribute, the server name and port number must match to the address at which your WASP for C++ server is listening (mine is listening on localhost:7777, for example). The server name and port number are followed by any directory that you want (for example, "wasp" in my location attribute). This directory entry doesn't have to exist, but it can be used by your server to differentiate between different Web Services. Finally, you have the method name being called (in my case that's the "jobadd" entry).
- The name attribute of the <port> element that's within the <server> element section should contain the service name. Also, that should match with the service that's implemented using the server-side WASP for C++ implementation. In my WSDL file it looked like this:

```
<port name='clsJobsSoapPort'
   binding='wsdlns:clsJobsSoapBinding' >
```

clsJobsSoapPort is the name of my service. WASP for C++ will use this setting to name your class that implements the service.

Generating

Once you have a working WSDL file, you can use the ServiceCompilerC that's part of the WASP for C++ download to generate the various stub files for you. The ServiceCompilerC is the compiler that WASP uses, and it's written in Java. The folks at Systinet have done this for convenience, because they'd already built lots of SOAP functionality into their Java libraries. The important point to know for any C++ developer is that at runtime, none of the Java packages are used.

Running the ServiceCompilerC is similar to running any other Java application. You need to specify arguments to it: the name of the WSDL file and a prefix for all of the generated files. In my case, I issued the following statement (also included in the file generateStubs.cmd, available in the Download file at www.xmldevelopernewsletter.com):

```
java -classpath
"C:\Program Files\wasp_cpp\libs\lib\log4j.jar;
C:\Program Files\wasp_cpp\libs\lib\wasp.jar;
C:\Program Files\wasp_cpp\libs\lib\xerces.jar"
com.idoox.soap.tools.ServiceCompilerC
wsJobBank.wsdl JobBank
```

Make sure that you update the classpath locations so that they match the location where you installed WASP for C++. The WSDL file is wsJobBank.wsdl, and the prefix selected is JobBank.

This command generates a number of files. Some of these files are important for the server-side implementation and some for the client-side implementation. For my service, the generated files were:

- JobBankIA.cpp and JobBankIA.h contain implementation adaptor classes capable of taking a SOAP request in a string, decoding it, passing the request data to the implementation of the service, encoding the return data, and returning the result message in a string. In short, this class converts SOAP requests to function calls and makes SOAP responses from the results.
- JobBankStructs.cpp and JobBankStructs.h contain helper structs for the data structures used by the service. These files are also used for the client-side implementation.
- ClsJobsSoapPortImpl.cpp and clsJobsSoapPortImpl.h are the skeletons of the implementation of the Web Service. These are named according to the port name in the WSDL file, which in this case is clsJobsSoapPort. The actual implementation can be written directly into this class. As an alternative, a new call could be created that's inherited from this implementation class.

Finally, you'll need to use the HTTP_SOAPServerHelper class for receiving requests and sending responses. One way to get started is to use one of the server implementations bundled with WASP for C++: All you have to do is modify the samples slightly to match your implementation.

In this example, I'll be using a stand-alone server implementation that doesn't support parallel message processing. If you need to use a powerful multithread server, you'll need to bind to your Web server, (for instance, IIS or Apache). This is outside of the scope of this article, but you can obtain more information on the topic from www.webware.at/SOAP/index.html, an area managed by Christian Aberger at WebWare.

You also need the Server.cpp file that's the actual simple Web server implementation. This file handles the incoming requests from a particular port on your machine. Currently this implementation handles only one Web Service, though it can be expended to handle more Web Service implementations, based on the URL submitted.

My Server.cpp begins with some includes:

```
#include <http/HTTP_SOAPServerHelper.h>
#include <soap/SOAP_Init.h>
#include <stdio.h>
// Each Web Service implementation
// needs something like the next two lines
#include "JobBankIA.h"
#include "clsJobsSoapPortImpl.h"
```

For each Web Service that this server is going to handle, you'd have to add include statements for the header files of the Implementation Adaptor (the "IA" ending) and the implementation of the Web Service (the "Impl" ending).

Next are some conditional includes and commands for handling signals in certain environments and being able to break when Ctrl-C is pressed:

```
#if !(defined (WIN32) || defined(_WIN32_WCE))
  #include <signal.h>
  #include <unistd.h>
  void handle(int) {
    SOAP_Terminate ();
    exit(0);
  }
#endif
```

Next is the main function of the Web server implementation:

```
int main(int argc, char **argv) {
  SOAP_Initialize ();
  #if !(defined (WIN32) || defined(_WIN32_WCE))
    signal(SIGINT, handle);
  #endif
  TRY(_exc) {
    JobBankIA ia;
    ia.setDefault_clsJobsSoapPortImpl_Instance
      (new clsJobsSoapPortImpl(), 1);
```

This code begins with a call to initialize SOAP, as well as signal processing conditionals for breaking out of the loop. Then, there's code to handle initiating the implementation adopter and pointing it to the particular implementation of the particular Web Service to match to the particular Web Service that you've implemented.

Next, I create the server that's going to handle the incoming requests and outgoing responses. The port number is set to 7777, but you can change it to match your environment. The port number can be changed when starting this Web server from the command line by providing a new port number as a parameter. For example, starting the service with this command line will cause the server to initialize itself and listen to port 8080:

```
JobBankServer.exe 8080
```

Back to Server.cpp code:

```
int i_Port = 7777;
if (argc > 1) {
  sscanf(argv[1],"%i",&i_Port);
} else {
  printf ("Can supply port as parameter\n");
}
HTTP_SOAPServerHelper server
  (i_Port, CONNPROTO_TCP, _exc);
CHECX_NESTED(_exc);
```

Once the server has been created, the following code will make the server go into an endless loop that waits for requests coming over HTTP on the port that you specified:

```
for (;;){
  printf ("Waiting for request on port");
  printf (" %i ...\n", i_Port);
  HTTP_SOAPServerHelper::request *req =
    server.getRequest(_exc);
  CHECX_NESTED(_exc);
  if (req == NULL) {
    printf ("Invalid request received!\n");
    continue;
  }
```

When a request is received, the code breaks out of the loop and in the next few lines, the URL is captured (it could be used to direct requests to different Web Services implementations). For my sample Job Bank Web Service article, there's only one Web Service that's being handled:

```
  char *tmp = req->url.transcode();
  printf ("url: %s\n",tmp);
  delete tmp;
  pxerces::DOMString result;
  int resultCode = ia.handleRequest
    (req->body, req->soapAction, result);
  server.respond
    (result, _exc, 500 - resultCode * 300);
  CHECX_NESTED(_exc);
  }
```

Once the Web Service is called, the results returned from the server are sent back to the client. The code ends with some exception handling code that also prints to the screen trace information if there's an error in the processing of the request or any other unknown error:

```
  } CATCH (SOAP_AdaptorException,e,_exc) {
      printf ("Adaptor exception: %s\n",
        e->getCharMessage ());
      char *tmp=GET_TRACE (e);
      printf ("Stack trace: %s\n",tmp);
      delete tmp;
      delete e;
  } AND_CATCH_ALL (e, _exc) {
      printf ("Unknown exception: %s\n",
        e->getCharMessage ());
      char *tmp=GET_TRACE (e);
      printf ("Stack trace: %s\n",tmp);
      delete tmp;
      delete e;
  } STOP_CATCH(_exc);
  SOAP_Terminate ();
  return 0;
}
```

That takes care of the server implementation code. My actual Web Service called from this server-side code is much simpler. This is the main portion of the clsJobsSoapPortImpl.cpp file:

```
#include "clsJobsSoapPortImpl.h"
#include <util/exceptions.h>
#if defined(WIN32) && defined(_DEBUG)
  #if defined(_AFXDLL)
    #define new DEBUG_NEW
    #undef THIS_FILE
    static char THIS_FILE[] = __FILE__;
  #endif
#endif
using namespace pxerces;
```

After the include directives and preprocessor instructions, there are the standard construction and destruction functions, where optional code could be placed.

```
clsJobsSoapPortImpl::clsJobsSoapPortImpl() {
}
clsJobsSoapPortImpl::~clsJobsSoapPortImpl() {
}
```

The actual implementation is contained within the following function. Of course, this sample implementation is very simple, and it returns a confirmation message composed from some of the submitted data. I've used the pxerces::DOMString class for handling strings used in the DOM C++ API:

```
pxerces::DOMString clsJobsSoapPortImpl::JobAdd
  (short _companyID,
  pxerces::DOMString _expireDate,
  pxerces::DOMString _jobTitle,
  pxerces::DOMString _jobDesc,
  double _salaryAmount) {
  if (_companyID ==1) {
    pxerces::DOMString salaryAmountStr =
      ((const SOAP_Primitive&)
        _salaryAmount).DOMStringValue();
    pxerces::DOMString strRes =
      "Your job with title " +
      _jobTitle + " and description " +
      _jobDesc + " was received " +
      "successfully on " + "2/6/2002" +
      ". The salary requested is " +
      salaryAmountStr + " and the expiration " +
      "date of this posting is " +
      _expireDate + ".";
    return (strRes);
  } else {
    return "Your cannot submit job postings.";
  }
}
```

Client-side implementation in WASP for C++

The client implementation is contained within the client.cpp file. Systinet provides a sample file in some of the demos that come with the WASP for C++ package. These can be helpful when creating your own clients. I have adapted this client to work with the JobBank Web Service from one of their examples:

```
#include <portable-dom/dom/dom/DOMString.hpp>
#include <dom/XMLDecoder.h>
#include <stdio.h>
#include "JobBank.h"

int main(int argc, char **argv) {
  SOAP_Initialize ();
  TRY(_exc) {
    printf ("Initializing stub...\n");
    clsJobsSoapPort server;
    if (argc>=2) {
      server._setAddress (argv[1]);
    } else {
      printf ("Can supply service URL ");
      printf ("as parameter\n");
    }
```

The main function of the WASP for C++ client Web Service starts by initializing SOAP and making a reference to the Web Service. Next, I specify the various parameters and their values. The parameter types aren't specified here, but are read from the WSDL file. At this level, all parameters are treated as strings, using the pxerces::DOMString class. Then, a call is placed to the server:

```
    short companyID_In = 1;
    pxerces::DOMString expireDate_In
      ("6/15/2002");
    pxerces::DOMString jobTitle_In
      ("Sr. Developer");
    pxerces::DOMString jobDesc_In
      ("Must have 10 years of exp...");
    double salary_In = 90000;
    pxerces::DOMString ds_Res = server.JobAdd
      (companyID_In,expireDate_In,jobTitle_In,
      jobDesc_In,salary_In,_exc);
    CHECX_NESTED(_exc);
```

Finally, in the following code section, the response from the server is retrieved and displayed on the screen:

```
    char *tmp=ds_Res.transcode ();
    printf ("Server: %s\n",tmp);
    delete tmp;
  } CATCH (SOAP_StubException,e,_exc) {
    // Error in communication
    printf ("Stub exception: %s\n");
    printf (e->getCharMessage ());
    char *tmp=GET_TRACE (e);
    printf ("Stack trace: %s\n",tmp);
    delete tmp;
    delete e;
  } AND_CATCH_ALL (e, _exc) {
    // Some unknown error
    printf ("Unknown exception: %s\n");
    printf (e->getCharMessage ());
    char *tmp=GET_TRACE (e);
    printf ("Stack trace: %s\n",tmp);
    delete tmp;
    delete e;
  } STOP_CATCH(_exc);
    // Terminate library
    SOAP_Terminate ();
    return 0;
}
```

New clients: ASP and VBScript

Now that I had the WASP for C++ Web Services server running, I wanted to connect to it using another language and environment. I selected ASP and VBScript as a client that wouldn't take much code—primarily because I'm running out of space in this issue.

So, here's the ASP client that calls the WASP for C++ Web Services server. This implementation uses the high-level SOAP API from the Microsoft SOAP Toolkit 2.0. Microsoft's SOAPClient was able to take the WASP for C++ WSDL file without any modifications to the WSDL. However, note that you may have to adjust your WSDL file path within the ASP code. To run this code, you'll have to place this ASP file under one of the IIS directories and browse to it in your browser:

```
<%
Dim soapClient
Set soapClient = _
  Server.CreateObject("MSSOAP.soapClient")
soapClient.mssoapinit _
  "C:\Program Files\wasp_cpp\jobBank" & _
  "\wsJobBank.wsdl"
Response.Write soapClient.JobAdd _
  (1, #12/31/2001#, _
  "Project Manager", _
  "Must have 15 years of experience...", _
  120000)
Set soapClient = Nothing
%>
```

That's all for this month. Throughout these few articles on connecting Web Services, I hope you were able to get a sense of what it takes to build Web Services on various environments and programming languages, and also become more familiar with some of the more common interoperability issues involved. Although there are challenges in connecting various Web Service implementations, the task is quite possible, even with the current versions of early generation Web Services implementations and toolkits. As time goes on, I expect to see vendors do more extensive interoperability tests, and many of the current issues will likely be addressed. Bottom line, just as with any other technology, when implementing Web Services you need to know some of the specific issues or difficulties that you ought to watch for, and the rest is no different than what you're used to doing when programming with any other tool or system. ▲

DOWNLOAD CWASPASP.ZIP at www.xmldevelopernewsletter.com

Zoran Zaev works as the principal software architect for xSynthesis LLC, a software and technology services company in the Washington, D.C. area. He enjoys helping others realize the potential of technology, and when he isn't working, he spends considerable time writing articles such as this one and books (he co-authored *Professional XML Web Services* and *Professional XML 2nd Edition* with Wrox Press). Zoran's research interests include complex systems that often involve XML, highly distributed architectures, and systems integration, as well as the application of these concepts in newer areas such as biotechnology. When not programming or thinking of exciting system architectures, you'll find Zoran traveling, reading, and exploring various learning opportunities. zzaev@yahoo.com.