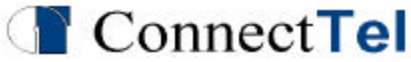

SOAP and Embedded Systems

Draft 0.3

May 11, 2001

by
Rosimildo Da Silva,
ConnectTel, Inc.
(rdasilva@connecttel.com)

1. Introduction

The increased demand for Internet connectivity among otherwise standalone devices, such as Gas Pumps, Industrial Controllers and MP3 players, means that embedded developers are frequently designing for connectivity. With a TCP/IP interface a must-have in many embedded processors, this paper provides reasons to use open standards, such as SOAP and XML, to expose the possible Web Services devices can provide.

Using open standards in developing embedded systems is a practical matter, since a number of p latforms currently exist for creating applications. Each of these platforms has traditionally used its own protocols, usually binary in nature, for machine-to-machine integration. As a result, applications across platforms have only a limited ability to share data. In recognizing these limitations, there has been an overwhelming push towards standards for data formats and for data exchange. This push stems from a vision that is rapidly evolving into a new computing paradigm, which is the seamless, Web-enabled integration of services across traditional hardware and software barriers.

This paper provides a simple example of how SOAP/WSDL can be used to make an embedded device a Web Service. A SOAP toolkit called Embedded SOAP Toolkit (eSoap), jointly developed by ConnectTel, Inc. and Technopoint International, is a SOAP library written in C++ and Java that is specifically targeted towards the development of embedded systems.

2. Web Services

Web Services combine the best aspects of component-based development and the Web. A Web Service is a unit of application logic providing data and services to other applications. Applications access Web Services via common Web protocols and data formats such as HTTP, XML, and SOAP, without requiring knowledge of how each Web Service is implemented.

The following link has a more detailed description of Web Services:
http://www.xml.com/pub/a/2001/04/04/webservices/index.html
http://www-106.ibm.com/developerworks/webservices/library/ws-peer1/

Usually embedded systems provide a *message-based protocol*, where a client sends messages to a server and receives the response back. To interface with a system, then, code must be written for the client application. This amount of specificity can be a problem if the inteface is complex (number of messages), and the formating (mashilling/unmashlling) of parameters are necessary.

Technically, this type of code could still be considered a Web Service, but it is usually easier to publish the device interface as a Web Service Description Language (WSDL) contract. WSDL is a language that lets you specify the interface of your service. It plays the same role as IDL plays to CORBA -aware systems. One interesting aspect of WSDL is that many tools vendors are adding WSDL parsers to their toolsets to automatically read the WSDL interface of your service, and generate the “proxy stubs” to talk to the service with all code being written by the tool.

Publishing the interface of your Web Service as a WSDL contract will make it very easy for “client applications” to talk to your device.

3. What is WSDL ?

The Web Services Description Language (WSDL) is an XML grammar for specifying properties of a Web Service, such as what it does, where it's located, and how to invoke methods (operations) on a particular service. Think of WSDL as the contract between the service and its clients.

Refer to the following link for more details:
http://www.w3.org/TR/wsdl.html

4. What is SOAP ?

SOAP stands for Simple Object Access Protocol. It is an XML-based, lightweight protocol for data exchange in a decentralized, distributed environment. SOAP request packets call methods available on the SOAP server, and the SOAP server sends a response packet, structured in a similar manner, back to the client. SOAP can be used with many transport protocols. The most common transport is HTTP.

This paper does not attempt to explain SOAP. For a complete description of SOAP, go to:
http://www.w3.org/TR/SOAP

SOAP has become the defacto standard protocol for Web Services. To expose the interface of an embedded systems as a Web Service, you should definitely be up to speed on SOAP.

5. An Embedded System as a Web Service

Now that we have given a brief introduction of SOAP and Web Services, let's apply these concepts to a typical embedded system.

What are the requirements to make an “embedded device” as a Web Service ?

- TCP/IP stack
- SOAP library (toolkit)
- 200K memory footprint
- C++ compiler
- RTOS with threads support(optional)

If you have a device with a standard TCP/IP stack implementation and a good C++ compiler, there are third-party SOAP libraries that can be used to build your Web Service more quickly than coding from scratch.

A specialized toolkit developed by ConnectTel, Inc, and Technopoint International, Embedded SOAP Toolkit (eSoap) is used in the next few sections. A SOAP library written in C++ and Java, eSOAP allows embedded systems developers to save time-to-market using COTS library for SOAP development.

See the following link for more information about eSoap:
http://www.embedding.net/eSOAP

6. Example Using eSoap

This section shows a sample program that calls a method called "add" with parameters "a" and "b" as integers. This call returns the sum of both numbers as an integer. The example is written in C++, and it is assumed that you have some understanding of XML.

Step 1: Write the WSDL file describing the interface of your service.

```
<?xml version = "1.0"?>
<definitions name = "ESoapServerService"
          targetNamespace = "http://localhost:8080/wsdl/esoapserver.wsdl"
          xmlns:xsd = "http://www.w3.org/1999/XMLSchema"
          xmlns:soap = "http://schemas.xmlsoap.org/wsdl/soap/"
          xmlns = "http://schemas.xmlsoap.org/wsdl/">
  <message name = "AddRequest">
       <part name = "a" type = "xsd:int"/>
       <part name = "b" type = "xsd:int"/>
  </message>
  <message name = "AddResponse">
       <part name = "sum" type = "xsd:int"/>
  </message>
  <portType name = "ESoapServerPort">
       <operation name = "add">
            <input  message = "AddRequest"  name = "addRequest"/>
            <output message = "AddResponse" name = "addResponse"/>
       </operation>
   </portType>
   <binding name = "ESoapServerBinding" type = "ESoapServerPort">
       <soap:binding style = "rpc" transport = "http://schemas.xmlsoap.org/soap/http"/>
       <operation name = "Add">
          <soap:operation soapAction=""/>
          <input>
             <soap:body use = "encoded" namespace = "urn:eSoap-Sample"
                  encodingStyle = "http://schemas.xmlsoap.org/soap/encoding/"/>
          </input>
          <output>
              <soap:body use = "encoded" namespace = "urn:eSoap-Sample"
                   encodingStyle = "http://schemas.xmlsoap.org/soap/encoding/"/>
          </output>
       </operation>
    </binding>
    <service name = "ESoapServerService">
         <documentation>Accepts two numbers and return the sum of them</documentation>
         <port name = "ESoapServerPort" binding = "ESoapServerBinding">
            <soap:address location = "http://localhost:8080/rpcrouter"/>
         </port>
    </service>
</definitions>
```

Step 2: write the server code (following section)

Step 3: write the client code (following section)

7. The Client

This section shows how to write a SOAP client application using the eSoap toolkit. This example is written in C++.

```
// STEP 1:  include the eSoap headers.
#include "soap_envelope.h"
#include "soap_transport.h"
```

```
// STEP 2:  endpoint to be called.
#define ESOAP_SERVER_URL    "http://localhost:8080/rpcrouter"

int main(int argc, char **argv)
{
  const char *url  =  ESOAP_SERVER_URL;

   // STEP 3:  create an Envelope instance to hold the request.
  esoap::Envelope    env;


   // STEP 4:  set method instance.
  esoap::Method *m = env.setMethod( "add" );


   // STEP 5:  add parameters for this call
  m->addInteger( "a", 100 );
  m->addInteger( "b", 20 );

   // STEP 6:  create the Transport instance.
  esoap::Transport *ht = esoap::TransportFactory::create( url, esoap::TransportFactory::HTTP );

   // STEP 7:  make the call
  esoap::Envelope *in = ht->call( env, "some:action" );

   // STEP 8:  check if response is OK
  if( in->success() )
  {
     // STEP 9a:  OK, show response
     esoap::Method    *resp   = in->getMethod();
     esoap::Parameter *sum  = resp->getParameter( 0 );
     printf( "Sucess: SUM is: %d\n", sum->getInteger() );
  }
  else
  {
     if( in->isFault() )
     {
       // STEP 9b:  Error, show what happened.
        esoap::Fault *f = in->getFault();
        printf( "Fault Received: Faultcode: %s\nFaultstring:%s\n",
                f->getCodeString().c_str(), f->getFaultString().c_str() );
     }
   }

   // STEP 10:  cleanup all instances.

  delete in;
  delete ht;
}
```

8. The Server

This section shows how to write a SOAP Server application using the eSoap toolkit. This example is written in C++.

// STEP 1: include headers.

```
#include "soap_server.h"
#include "soap_envelope.h"

// Default Port Number and Config file
#define ESOAP_PORT  8080
#define ESOAP_CONF  "esoapserver.conf"
```

// STEP 1: define “add” method

```
/**
 * The SOAP server framework call this method
 * to add two parameters.
 */
int methodAdd( esoap::MethodDataBlock *mdata )
{
  esoap::Parameter *pa;
  esoap::Parameter *pb;

  esoap::Method *m = mdata->in()->getMethod();
  pa = m->getParameter( "a" );
  pb = m->getParameter( "b" );
  if( pa && pb )
  {
    esoap::String respName = "returnResponse";
    esoap::Method *resp = mdata->out()->setMethod( respName.c_str() );
    int res = pa->getInteger() + pb->getInteger();
    resp->addInteger( "sum", res );
  }
  else
  {
    printf( "ERROR: parameter not found\n" );
    esoap::Fault *resp = mdata->out()->setFault(
               esoap::Fault::Server, "Parameters are missing” );
  }
  return 0;
}
```

// STEP 3: Server example using Abyss.

```
int main(int, char ** )
{
  // Create server instance
  esoap::ServerFactory::create( esoap::ServerFactory::ABYSS );
  // Init server...
  esoap::Server::instance()->init( "Server", ESOAP_PORT, ESOAP_CONF );
  // register method "add" with the callback above.
  esoap::Server::instance()->getRegistry().addMethod( "add",
          new FunctionCallback< esoap::MethodDataBlock >( methodAdd ) );
  // run the server...
  esoap::Server::instance()->run();
  return 0;
}
```

9. Conclusions

As you could see, it is quite simple to add a standard Web Service interface to your device. Using third-party libraries such as eSoap can speed up development time of your embedded system , thus improving the time to market for your system.

One of the major advantages of using a third-party library like eSoap is that the library hides all aspects of SOAP/HTTP. In order to write a code that makes a Web Service device,developers do not need to become experts in SOAP. Instead, they can now completely focus on the domain-specific details of the application. Developers need only become familiar with a few classes of the SOAP toolkit to have the embedded system connect to the Web quickly.

For questions or comments you can contact the author (rdasilva@connecttel.com)

Rosimildo Da Silva is *CTO and co-founder of ConnectTel, Inc., an Austin (Texas)-based software development firm, specializing in designing and implementing firmware and drivers for embedded systems. Da Silva is specifically noted for working with software applications that interface with hardware components. He has extensive experience as the software architect on various projects and has also served as the lead architect on global projects.*