Reprint from the January 2002 Web Services Developer's Journal:

Wireless Web Services with J2ME Part II – SOAP without the rope

by Kyle Gabhart & Jason Gordon

Last month, we discussed J2ME and accessing web services from wireless devices using the XML-RPC protocol. In this second article, we will consider SOAP as a vehicle for accessing web services from wireless devices and compare and contrast it with XML-RPC. Our sample application will again be a J2ME midlet, however we will use EnhydraME's kSOAP rather than kXML-RPC to provide the protocol's implementation.

Overview of SOAP

The **S**imple **O**bject **A**ccess **P**rotocol is, according the 1.1 specification, "a lightweight protocol for exchange of information in a decentralized, distributed environment." The protocol is entirely based upon XML, is vendor-neutral, and is one of the cornerstone technologies in the web-services revolution. It is quite similar to XML-RPC, but we will examine this more closely in the next section.

SOAP was originally conceived and initially developed at Microsoft between 1998 and 1999. SOAP did not, however, gain widespread attention until DevelopMentor, IBM, Lotus, and Microsoft submitted the SOAP 1.1 version to the W3C on April 26, 2000. With both IBM and Microsoft behind it, the industry began to give SOAP some serious consideration. As of this writing, the SOAP 1.2 specification is currently being drafted by the W3C's XML Protocol Working Group. The latest draft of the 1.2 spec can be found at http://www.w3.org/TR/soap12/.

When examining SOAP, it is important to identify the three main components of any SOAP message: **the SOAP envelope**, **rules for encoding data**, and a **request/response interaction mechanism**. The SOAP messaging architecture can be compared to a postal system. A document is enclosed in an envelope and that envelope is transmitted via the transport mechanism, the mail system in our analogy, or HTTP across a network in the case of SOAP. With the mail, a zip code identifies the locale that a envelope should be delivered to, with SOAP, HTTP header data provides such routing information. At this point, the analogy breaks down. Whereas the "last mile service" provided by a postal carrier is directed by the street address on the envelope, it is information encoded in the body of the XML document itself that ultimately directs the SOAP request to the specific service on the server.

XML-RPC vs SOAP

Although SOAP and XML-RPC have similar roots (XML-RPC is based on a subset of the original SOAP spec that was developed at Microsoft in the late 1990s), they are very different animals. Both XML-RPC and SOAP are XML-based protocols for communication and data exchange, so when should one be chosen over the other? We'll take a look at their strengths and weaknesses, and then evaluate these two protocols from a business perspective.

XML-RPC is an extremely lightweight mechanism for invoking XML-based services. It is a clean, simple protocol that provides the minimum overhead necessary to invoke remote services and exchange data in a platform, language, and vendor-neutral manner. It defines six simple data types and two complex types. The result is that XML-RPC is a highly-efficient lightweight protocol. Messages are simple to construct, simple to parse, simple to debug, and are easily human-readable. XML-RPC requires a minimal amount of active memory for processing building and parsing messages, and produces a thin message body to be exchanged between client and server.

In contrast, SOAP is a fatter protocol (although generally considered lightweight). In exchange for some additional overhead, SOAP provides namespace awareness, a sophisticated data-typing mechanism, and a flexible messaging paradigm. The W3C's XML Namespaces specification is leveraged to provide

namespace awareness, and the XML Schema specification provides the data-typing mechanism. As such, SOAP supports over 40 standard data types and provides the capabilities to define custom simple and complex data types. This provides a tremendous degree of flexibility in terms of describing robust data structures with intricate relationships with other data contained in the message body. The SOAP architecture also introduces a flexible messaging architecture, supporting a variety of messaging paradigms, including unidirectional, bi-directional, multi-cast (publish-subscribe), and sequential messaging (multiple parties chained together in a particular order). One aspect that facilitates these paradigms is the ability for a SOAP message to include a header section. This allows security, transaction, and routing information to be exchanged between multiple parties. For example, a client can send a SOAP request with an intended destination, but also declare in the SOAP envelope's headers that a particular party should receive the message, process the pertinent header information and then pass the message on to the next party in the chain (or the intended recipient if no other parties have been declared). One final distinction is that SOAP supports asynchronous messaging, while XML-RPC requires a synchronous communication between the two parties in an exchange.

With all of these differences between the two protocols, executives, project managers, and wireless architects alike are wondering – "How do I make a business decision between SOAP and XML-RPC for a particular wireless project or system?" Well, even if you weren't wondering that, we're going to tell you anyway. In a nutshell, XML-RPC provides a fast, compact protocol for exchanging data and invoking services in a neutral, standardized way. If application size, memory, and bandwidth are your top priorities, then XML-RPC is the way to go. On the other hand, if your application requires a robust data-typing mechanism, extensive security or transaction support, or a flexible messaging architecture, than SOAP is your answer. Also, SOAP is more mainstream than XML-RPC, so if your application needs to access a variety of services that you have no control over, SOAP may be a better match from an interoperability perspective. To break it down into more detail, we have provided a table (figure 1) that outlines the business justifications for using each protocol.

Business Priority	**Appropriate Protocol**
tiny memory footprint	XML-RPC
speed and efficiency	XML-RPC
interoperability with other parties beyond influence or control	SOAP
reduced bandwidth	XML-RPC
easy maintenance and debugging of system	XML-RPC
exchange of robust data structures with intricate relationships	SOAP
flexible messaging architecture	SOAP
creation of named, custom data structures	SOAP
creation of simple, non-named custom data structures	XML-RPC
asynchronous communication	SOAP
additional security beyond HTTPS and SSL3	SOAP
transaction support	SOAP

Figure 1: The decision between XML-RPC and SOAP should be based upon the requirements of your application.

Wireless SOAP Example

In our previous article, we developed a sample application using kXML-RPC (http://kxmlrpc.enhydra.org) an open-source implementation of the XML-RPC protocol for micro devices, maintained by Enhydra. kSOAP (http://ksoap.enhydra.org) is another Enhydra Micro Edition project, providing SOAP support for mobile devices. We will create a MIDP interface with the kSOAP libraries underneath to activate a SOAP-based web service located on a remote HTTP server.

For our application, we will be using three classes from the kSOAP library to handle the marshalling and unmarshalling of data via SOAP:

- `HttpTransport` – a convenient API to enable SOAP calls via HTTP using the J2ME Generic Connection Framework
- `ClassMap` – provides namespace support and a two-way mapping between namespace-qualified XML names and Java classes
- `SoapObject` – a generic object used to represent any SOAP object within the body of a SOAP request or response. SOAP objects can also be nested.

Rather than creating both the client and the service we are accessing, we will simply access an existing service. Xmethods.com provides a web service registry for publicly available web services. We will be invoking a service provided by CapeClear that gathers and disseminates airport weather information on behalf of a client. The details regarding that service can be located at: [l?id=129](#) and the source code for this midlet (AirportWeather.java) ebsite at http://ksoap.enhydra.org/software/downloads/index.html .

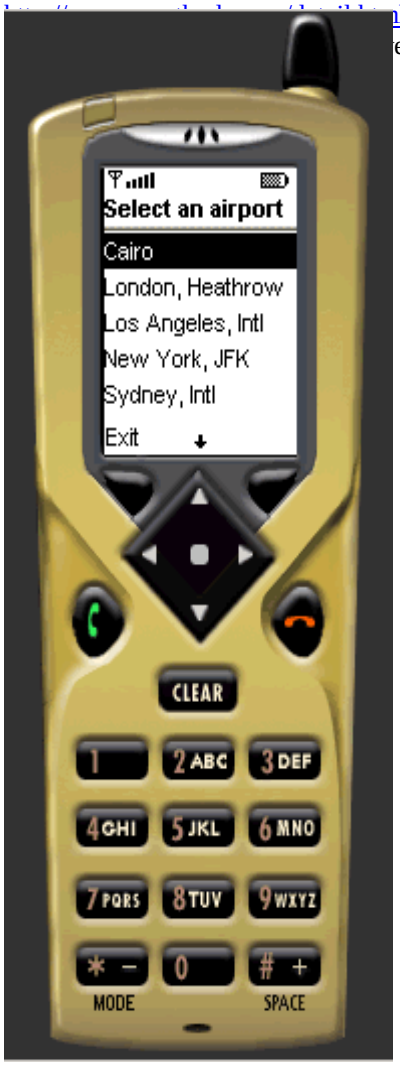

Figure 2

In creating our sample SOAP midlet (AirportWeather.java), the process is identical to the one we used in our previous article. This time, the packages and names have been changed to protect the innocent. We will import and extend and use classes from the kSOAP library rather than the kXML-RPC library. The first step obviously, is to import the necessary packages and declare the MIDP components that will be used in the application. After this, we define the midlet's constructor, initializing all the UI components, and add them to the display as necessary. With that complete, we need to fill in the three other lifecycle methods. In the startApp() method, we simply bring the MIDP display into action. Since we don't use any shared resources, the pauseApp() method is blank. Finally, the destroyApp() method releases the local resources that we have allocated for our midlet.

As with our previous article, all the action takes place in the commandAction() method. This method is called anytime the user performs a command event (pressing a key, selecting an item from a list, etc.). The Command and Displayable objects are then queried to determine which component has actually been activated/deactivated, and the appropriate actions are performed. When our midlet is launched, it displays a list of popular international airports for which weather information can be retrieved (see Figure 2).

It also displays an exit button to allow the user to exit the midlet. Upon selecting an airport, a list of weather information services for that airport are displayed (see Figure 3).

A back button is also displayed to allow the user to return to the airport menu. The user then selects the weather information service they are interested in and the midlet sends a SOAP request to the server to collect the desired information. A SOAP response is returned, parsed by the midlet, and the result is displayed on the screen. So how does all of that work? The commandAction() method.

```
public void commandAction( Command com,
                           Displayable dis ){

  if ( dis == airportMenu && com ==
     List.SELECT_COMMAND ) {
      // An airport has been selected
  }
  else if ( dis == servicesMenu && com ==
     List.SELECT_COMMAND ) {
      // A weather service has been selected
  }
  else if ( com == backCommand ) {
      // The back button returns to the
      // airport list screen
  }
  else if ( com == exitCommand ) {
      // The exit button exits the midlet
  }//end if ( dis == airportMenu && com ==
List.SELECT_COMMAND )
}//end commandAction( Command, Displayable )
```

Figure 3

The outer level of the commandAction() method checks to see what component has been activated. We'll look at each of these four events individually, beginning with the airport menu:

```
  if ( dis == airportMenu && com == List.SELECT_COMMAND ) {
    currentAirport = airportMenu.getSelectedIndex();
    servicesMenu.setTitle( airports[ currentAirport ] + " weather" );
    display.setCurrent( servicesMenu ); //display the list of services
```

When an airport is selected, we set the currentAirport variable, set the title for the services menu to reflect the selected airport, and display the services menu.

Next we'll look at the services menu. There are seven possible weather services that can be retrieved: wind conditions, temperature, pressure, humidity, sky conditions, visibility, and a complete summary containing the six individual pieces of information:

```
  else if ( dis == servicesMenu && com == List.SELECT_COMMAND ) {
    String result = null;
    int choice = servicesMenu.getSelectedIndex();
```

The first thing to do is to declare a String variable to store the result of the service call that will be made. Next, determine which service has been selected and store the index value in an integer variable and perform a switch on that variable. The switch statement can be seen in Listing 1. The first case retrieves a

summary and the other cases retrieve individual weather items. After the switch statement, the results are displayed on the screen via an Alert object.

Within each case statement in the switch block, the callService() method is used to actually handle the exchange of SOAP messages.

```
private Object callService( int choice, String methodName ) {
  Object result = null;

    try {
      transport = new HttpTransport( serviceUrl, soapAction + "#" +
            methodName );
      transport.debug = true;

      classMap = new ClassMap();
      classMap.prefixMap = new PrefixMap( classMap.prefixMap, "air",
            serviceNamespace );
      transport.setClassMap( classMap );

      request = new SoapObject( serviceNamespace, methodName );
      request.addProperty( "arg0", airportCodes[ choice ] );
      result = transport.call( request );
    } catch( Exception e ) {
      e.printStackTrace();
      System.out.println( "Request: \n" + transport.requestDump );
      System.out.println( "Response: \n" + transport.responseDump );
      result = null;
    }//end try/catch

    return result;
  }//end callService()
```

Four essential things take place in this method. An HttpTransport object is created, a PrefixMap namespace is created for the SOAP envelope, a SoapObject is created to represent the request, and the request object is sent via the HttpTransport class's call() method.

The next else/if block checks to see if the back command has been selected, in which case the list of airports is displayed:

```
      else if ( com == backCommand ) {
        display.setCurrent( airportMenu );
```

Finally, the exit command is checked to allow the application to be exited:

```
      else if ( com == exitCommand ) {
        destroyApp( true );
        notifyDestroyed();
```

That's all there is to it. Download the source code to get the details for the interface and then you are ready to build the midlet interface, compile the code and access this weather service from a J2ME phone using SOAP!

Conclusion

Web services are sweeping the industry and changing the face of business. Mobile computing and wireless access to information at any time, from anywhere, is an increasingly popular idea. The combination of the two is explosive! In these two articles, we've explored XML-RPC and SOAP as protocols for accessing web services from wireless devices. XML-RPC provides a highly-efficient, extremely lightweight mechanism for invoking web services that is ideal for mobile and embedded devices. SOAP provides a slightly heavier protocol with increased functionality and flexibility. Between these two, an appropriate protocol should be able to be found for any wireless project.

Listing 1:

```
        switch( choice )
        {
          case 0:
            SoapObject objResult = (SoapObject) callService(
currentAirport, "getSummary" ); //service call
            if ( objResult != null )
            {
              result = "The weather at " + objResult.getProperty(0) +
                " is " + objResult.getProperty(3) + " with a " +
                objResult.getProperty(2) + " sky, and wind " +
                objResult.getProperty(1) + ". The humidity is " +
                objResult.getProperty(4) + ", the presssure is " +
                objResult.getProperty(5) + ", and the visibility is " +
                objResult.getProperty(6) + ".";
            }//end if ( objResult != null )
            break;
          case 1:
            result = (String) callService( currentAirport,
"getHumidity" ); //service call
            break;
          case 2:
            result = (String) callService( currentAirport,
"getPressure" ); //service call
            break;
          case 3:
            result = (String) callService( currentAirport,
"getSkyConditions" ); //service call
            break;
          case 4:
            result = (String) callService( currentAirport,
"getTemperature" ); //service call
            break;
          case 5:
            result = (String) callService( currentAirport, "getOb" );
//service call
            break;
          case 6:
            result = (String) callService( currentAirport, "getWind" );
//service call
            break;
        }//end switch( servicesMenu.getSelectedIndex() );

        //display the result
        response.setString( result );
        display.setCurrent( response );
```