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

Wireless Web Services with J2ME – Part 1

by Kyle Gabhart & Jason Gordon

What happens when the hype of web services, meets the increasingly popular and ever-changing world of wireless computing? Most likely, confusion and disillusionment. In this two-part series of articles, we will explore the uncharted waters of wireless web services. We will use the J2ME platform for developing our web service clients and access remote services on the Internet using standardized industry protocols. In this first article, we will examine XML-RPC, a simple, lightweight mechanism for invoking remote services with XML. The second article will compare and contrast XML-RPC with SOAP, a more robust, sophisticated, and heavier solution for invoking remote services with XML.

The Wireless World

A web service is a coarse-grained interface to one or more business services that is invocable across a network. With a wireless network, this invocation process becomes more complicated. Many cellular telephone service providers use analog circuit-switched networks that open a constant connection for the duration of the exchange. More advanced providers are moving to digital packet-switched networks. In packet switching, a stream of digital bits is carved up into bit clusters, called packets, and blasted across the network individually.

Circuit-switched analog networks are more expensive to maintain and offer limited bandwidth. Digital packet-switched networks are cheaper, more efficient, and do not have the same bandwidth limitations. The tradeoff with packet-switched networks is that packets are occasionally lost ('dropped') during transmission. Dropped packets must be retransmitted. The larger a transmission is, the greater the likelihood that packets will be dropped, requiring retransmission and degrading performance.

The bottom line is that regardless of the type of network being used by a provider, wireless clients must keep their exchanges as thin as possible to ensure optimum performance. Additionally, mobile devices typically do not have an abundance of resources for processing fat requests, responses, or storing robust data models.

XML-RPC for Wireless Web Services

XML-RPC is a Remote Procedure Calling protocol that invokes remote procedures over a network by sending XML-formatted messages. The XML-RPC specification was developed and is maintained by Userland Software, Inc and the full specification can be found at *http://www.xmlrpc.org/spec*.

XML-RPC is an extremely lightweight mechanism that can be used as a part of a web services architecture. The key to a web services architecture is the utilization of XML as a language-agnostic, vendor and platform-neutral medium for accessing Internet or intranet services. XML-RPC provides the minimum functionality necessary to specify data types, pass parameters, and invoke remote procedures in a neutral way.

What makes XML-RPC so efficient? XML-RPC defines 8 data types, six primitive types (int, Boolean, string, double, datetime, base64) and two complex types (struct and array). These are the only types available, yet they provide all the functionality that is needed about 80% of the time. Although SOAP provides a more robust data typing mechanism based upon XML Schemas (even allowing the creation of custom data types), this is often overkill in a wireless environment. We'll explore these topics more in next month's article, for now, we simply need to understand that XML-RPC is an extremely lightweight mechanism for invoking web services in a standardized and neutral manner.

The wireless applications that we will be developing in this article and the next one require the J2ME platform, so we will take a brief look at J2ME to provide for a basic background for these wireless web services.

J2ME Primer

The Java 2 Micro Edition is a Java 2 platform for developing applications for devices with limited memory. Specifically, J2ME addresses the need for application development for consumer and embedded devices. Because J2ME is designed for devices with extremely small footprints, many of the features of the J2SE are not included. Some of the notable features not included are floating point data types, serialization (no JavaBeans), thread groups and thread daemons, finalizations, user-defined class loaders and the JNI. As figure 1 indicates, the J2ME platform is a layered stack consisting of a virtual machine, the core J2ME class libraries, as well as configurations and profiles.

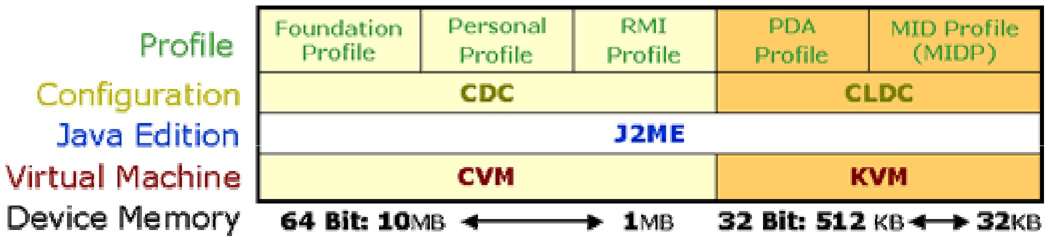


Figure 1: The J2ME Stack consists of a virtual machine, the standard J2ME libraries, configuration class libraries, and a device-specific profile.

Configurations

Configurations define the run-time environment by specifying the Java features (classes) that are available as well as which virtual machine will be used. A configuration can also be thought of as relating to a category of devices that have common characteristics and memory constraints. For Devices that have a total memory from 160 to 512 kB, the Connected Limited Device Configuration (CLDC) for J2ME can be used. CLDC devices usually include cell phones, two-way pagers and low-end PDAs. The CLDC also targets devices with a network connection and processing power of 16 or 32 bits. The CLDC uses the K (k for kilobyte) Virtual Machine or KVM. For Devices that have a total memory of 2MB or greater and have a 32-bit or 64-bit microprocessor, the Connected Device Configuration (CDC) is used. The CDC uses the CVM and is generally used on set-top TV's, higher-end PDA's, and next generation mobile devices. A configuration (and corresponding virtual machine), combined with a device-specific profile, and the core J2ME libraries, constitutes a complete J2ME environment.

Profiles Overview

Profiles work on top of configurations and focus on a "vertical" market or Industry segment of devices. Profiles also allow developers to address more device-specific features such as the life cycle of an application, user interfaces and networking issues. CDC devices typically use the Foundation profile, which targets devices that require more networking capabilities and no GUI's. CLDC devices typically use the Mobile Information Device Profile (MIDP). For the wireless development that we will be focusing on in this and the next article, we will be using MIDP.

MIDP

The MIDP consists of APIs for user interface design as well as for database activity. A MIDP application is referred to as a "midlet". MIDP even allows multiple midlets to be packaged together as a midlet suite and share information between midlets within the suite. This is generally only useful however, in the case of midlets that maintain a database. For our purposes, we are interested in MIDP's GUI capabilities. MIDP supports 10 GUI components: Command, Alert, Choice, Choice Group, Form, List, StringItem, TextBox, TextField and Ticker.

In our sample application, we will be using the following GUI components:

- **List** – Contains a list of choices, typically relies upon a device's "select" or "go" functionality
- **Command –** Presents a choice of action. Contains a label, a type and a priority.
- **Display –** The midlet's canvas upon which UI components are displayed.
- **Alert –** Informs the user about an exceptional condition or error. It can also be used to display the results of a query to the user.

To understand how a MIDP user interface is created and how it functions, it is necessary to understand the lifecycle of a midlet. The lifecycle of a midlet can be seen in four stages, each with a corresponding method defined within the midlet:

- *Initialization:* **constructor** – Every midlet has a default constructor. This is used to initialize a midlet's data members, including GUI components, with their desired property values (size, shape, color, label, text, reference, etc.)
- *Activiation:* **startApp()** – Acquires necessary resources, makes display visible to user, and begins to perform requested services.
- *Passivation***:** **pauseApp()** – Stops performing services and releases shared resources
- *Destruction:* **destroyApp()** – Releases shared and local resources and saves any persistent data.

The mobile device will handle the management of a midlet through these lifecycle methods via Application Management Software (AMS). AMS frees the developer from directly managing a midlet and its resources.

Writing a MIDP Web Service Client

In this article, we will be creating a MIDP client that uses the XML-RPC protocol for invoking remote web services in a platform and language neutral way. To do this, we will need a J2ME implementation of the XML-RPC protocol. As of the time of this writing, the only publicly available client implementation is kxmlrpc, an open-source XML-RPC project for the J2ME platform. Kxmlrpc is maintained by the Enhydra organization and can be freely downloaded from their website at *http://kxmlrpc.enhydra.org/*. The kxmlrpc library uses enhydra's kxml parser to handle the low-level XML parsing details. With the addition of the parser, the kxmlrpc jar file reaches a whopping 30k!

With the kxmlrpc jar file downloaded to your system and placed in your application classpath, you can set about writing the MIDP client. We'll walk through the creation of the midlet and highlight the most interesting lines of code, but the entire source code for the midlet can be seen in listing 1, and the source for MyMidlet.java can be downloaded from the kxmlrpc website (kxmlrpc-samples.zip) located at *http://kxmlrpc.enhydra.org/software/downloads/index.html*.

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 adding 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.

Now we are ready for the interesting part of the code, 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. Our midlet has three services displayed in a list (see Figure 2), and a switch statement is performed on the index of that list to determine which item has been selected. In listing 1, only the first service is given an implementation, but the other two can be seen by downloading the source code.

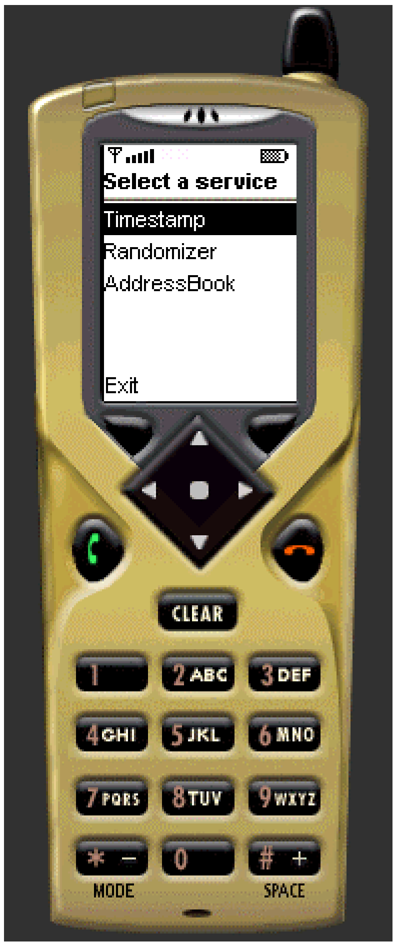
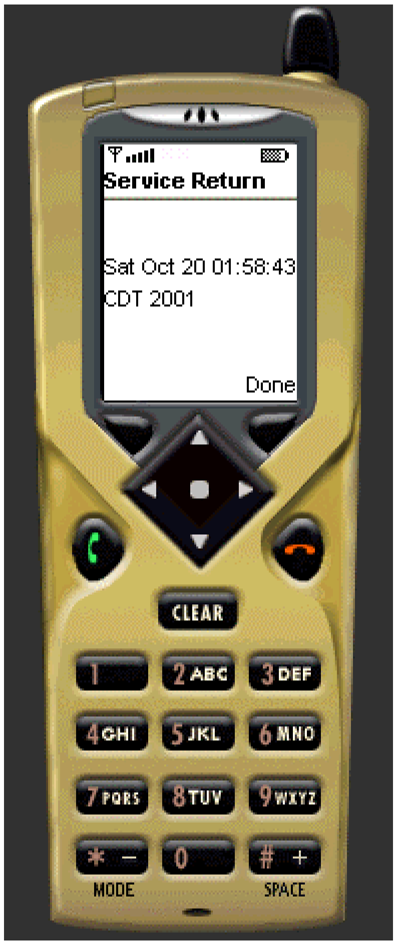


Figure 2: This XML-RPC demo midlet provides client access to three remote XML-RPC web services.

The Timestamp service is very simple, a parameter-less request is sent to the service and a String object representing the current time is returned. To perform this query, a KxmlRpc object is created with the specified URL for the web service. Then an empty Vector is created and the actual request is performed with the following line:

```
String serverTime = ( String ) xmlrpc.execute( "sysTime.getSystemTime", params );
```

The execute() method accepts two parameters, a String representing the name of the service, and a Vector representing any parameters that should be passed to the service. This particular service returns a String object that is then sent to the screen to display the current time on the server. An example can be seen in Figure 3.


Service Return

Sat Oct 20 01:58:43 CDT 2001

Done

Figure 3: The Timestamp service returns a string representation of the server's current date and time and displays this value on this screen.

Deploying and Testing a Midlet

For deploying and testing our midlet, we used Sun's J2ME Wireless Toolkit (J2MEWTK) version 1.0.3 beta which can be downloaded from Sun's website at *http://java.sun.com/products/j2mewtoolkit/*. The toolkit is 100% Java, built using the Java 2 Standard Edition, so even though it contains the J2ME APIs and is used for deploying and testing J2ME applications, it requires a J2SE implementation in order to run.

In order to deploy and test your midlet, you need to do four things: create a project for your application, write the midlet's code, place all of the files and resources in their appropriate application directories, and then build and run the application.

With the toolkit properly installed, the first step is to create a project for your application. To do so, follow these steps:

- Start the KToolbar application
- Click the "New Project" button. Name your project and name your project's midlet (This will also be the name that use for the midlet in the .java source file).
- Click the "OK" button on the Settings screen that shows keys and values. This screen represents your application's deployment properties. You can specify these properties now, or later by clicking the "Settings" button.

Now that you've created a project, the toolkit has created a corresponding directory structure for your project. That project directory structure is located under the apps directory of the J2MEWTK installation directory. We are only interested in three of them: the source code, resource, and library directories.

- **\src** – Place your java midlet's source code files in this directory
- **\res** – Place any resource files (images, text files, etc.) in this directory
- **\lib** – Place jar files and Java class files that your midlet(s) will need in this directory

After creating a project, writing the midlet code, and placing all the necessary files in the appropriate directories, you are ready to actually test the application. This requires three essential steps:

- Build the application into an executable midlet by clicking the "Build" button
- Resolve any errors or exceptions that are thrown and re-build the application until a successful build is accomplished
- Execute the application by clicking the "Run" button after a successful build has been created

When you run a midlet from the J2MEWTK, a phone emulator is started and the toolkit attempts to load your midlet into the emulator. You can test your midlet with any of the supplied emulators, or even download additional emulation environments from the web. With your midlet running, you can navigate through your midlet just as you would on a real J2ME-enabled wireless phone. If your computer is currently connected to the web, then you should be able to access the services listed in your midlet code.

Deploying into Production

Once your midlet development and testing is complete, you can package your application into an executable format by selecting the "Package" menu item from the "Project" menu. The toolkit will create a .jar and .jad file in your project's \bin directory. The .jad file is used for describing and executing your midlet, while the .jar file contains the Java class files, library and resource files used by your midlet. From now on, simply double-clicking the jad file will run the midlet.

Looking Ahead

In this first article we've taken a look into the world of the wireless web, XML-RPC as a web service communication protocol, the J2ME environment with special attention paid to the MID profile, and also taken a look at a demonstration of an XML-RPC midlet using the kxmlrpc code. XML-RPC provides a very thin, efficient means of invoking remote services in a standard and neutral way. It defines a succinct set of 8 data types, providing the means necessary to encode simple and moderately complex data structures in a highly efficient manner. More often than not, XML-RPC will provide you with all the functionality that you need, especially given the natural constraints of wireless devices. For applications that require more functionality, the Simple Object Access Protocol (SOAP) may be in order. Next month we will delve into SOAP and provide a detailed analysis of when to choose SOAP over XML-RPC for wireless computing.

Listing 1:

```
import javax.microedition.midlet.*;
import javax.microedition.lcdui.*;
import javax.microedition.io.*;
import java.io.*;
import java.util.*;
import org.kxmlrpc.*;

public class MyMidlet extends MIDlet
            implements CommandListener {

  private List            list;
  private Command         exitCommand;
  private String[ ]       menuItems;
  private Display         display;
  private Alert           response;
private XmlRpcClient   xmlrpc;
private Vector               params, xmlArray;

public MyMidlet() {
    //Initialize the User Interface
    menuItems = new String[ ] {"Timestamp",
                   "Randomizer", "AddressBook"};
    list = new List( "Select a service",
                 List.IMPLICIT, menuItems, null );
    exitCommand = new Command( "Exit",
                  Command.EXIT, 1 );
    response = new Alert("Service Return",
                   null, null, AlertType.INFO);
    response.setTimeout( Alert.FOREVER );

    //Add commands
    list.addCommand( exitCommand );
    list.setCommandListener( this );

    //obtain a reference to the device's UI
    display = Display.getDisplay( this );
}//end MyMidlet()

public void startApp() {
    display.setCurrent( list );
}//end startApp()

public void pauseApp() {
}//end pauseApp()

public void destroyApp( boolean bool ) {
    //clean up
    list = null;
    exitCommand = null;
    display = null;
}//end destroyApp

public void commandAction(
                Command com, Displayable dis ) {
  if ( dis == list &&
     com == List.SELECT_COMMAND ) {
     switch( list.getSelectedIndex() )
     {
         case 0:
           try
           {
               xmlrpc = new XmlRpcClient( "http://
         www.wsjug.org/servlet/XmlRpcServlet" );
                params = new Vector();
                String serverTime = (String) xmlrpc.
   execute( "sysTime.getSystemTime", params );
                response.setString(
                                       serverTime.toString() );
                display.setCurrent( response );
           }
           catch ( Exception ex ) {
                 response.setString( ex.toString() );
                ex.printStackTrace(); // DEBUG
                display.setCurrent( response );
            }//end try/catch
            break;
          case 1:
              response.setString(
                        “Please download the full sample code" );
              display.setCurrent( response );
              break;
          case 2:
              response.setString(
                        " Please download the full sample code" );
              display.setCurrent( response );
              break;
         }//end switch( list.getSelectedIndex() )
     }
     else if ( com == exitCommand ) {
        destroyApp(true);
        notifyDestroyed();
     }//end if ( dis == list && com == List.SELECT_COMMAND )
   }//end CommandAction( Command, Displayable )
}//end MyMidlet
```