Chapter B1. An Echoing Client/Server Application Using Bluetooth

Bluetooth is a wireless technology for communication over distances of up to 10m, offering reasonably fast data transfer rates of around 1 Mb/s, principally between battery-powered devices. Bluetooth's primary intent is to support the creation of ad-hoc personal area networks (PANs) for small data transfers (or voice communication) between devices such as phones and PDAs.

Bluetooth is ideal as the communications layer for mobile games involving a small number of players, such as sports (e.g. tennis) and networked board games (e.g. Go). An oft-cited drawback that the players must be physically near each other is actually perceived as an advantage by people who enjoy a social element in their gaming.

Bluetooth-based games offer a playing experience quite different from traditional networked gaming (e.g. MMORPGs) with their emphasis on size and interactions-at-a-distance. Those types of mobile games will most likely utilize 802.11b (WiFi) technology, acting as an extension to existing (non-wireless) networks, such as LANs.

Bluetooth PANs come in two main configurations: *piconets* and *scatternets*. A piconet is analogous to a client/server application, and is the focus of this chapter. Piconets are so named because they only permit a maximum of seven clients to be connected to a server at a time.

A scatternet lets a server be the client of another piconet, enabling the construction of peer-to-peer networks. Scatternet functionality is not widely supported on phones and PDAs at the moment.

The principal aim of this chapter is to introduce J2ME's Bluetooth API (known as JSR 82) by explaining how to use it in a simple client/server application. (At the Bluetooth networking level, servers are usually called *masters*, and clients are *slaves*; I'll forgo that naming scheme here, sticking to the more usual client/server terms.)

This chapter's example shows how a server 'registers' itself as a Bluetooth server, and processes client connections and messages. A client searches for Bluetooth devices and services, connects to a matching server, and sends messages to it. Several clients can be connected to the server at once, with the server using a dedicated thread for each one. Many of the Bluetooth-related classes developed here will be used in later chapters.

Once communication has been established, the application task is quite simple: a message sent by a client is echoed back by the server thread, changed to uppercase.

Figure 1 shows the main client and server screens in the application:

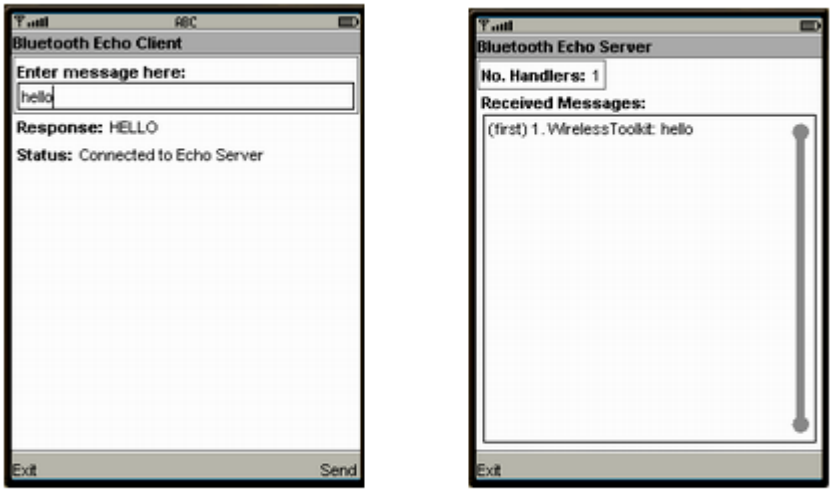

Figure 1 The Bluetooth-based Echo Client and Server.

The client has just sent the message "hello", and received back "HELLO". The server shows the number of client handlers it is currently managing (only one), and displays a list of messages received from its clients.

The clients and server MIDlets were tested using Sun's WTK 2.2, with everything running on the same machine. The MIDlets use JSR 82's Bluetooth API, which is offered as an optional component in the WTK.

1. JSR 82 Details

JSR 82 is a Java layer over a Bluetooth software stack which hides the underlying Bluetooth hardware or firmware in the device. Figure 2 shows these layers:

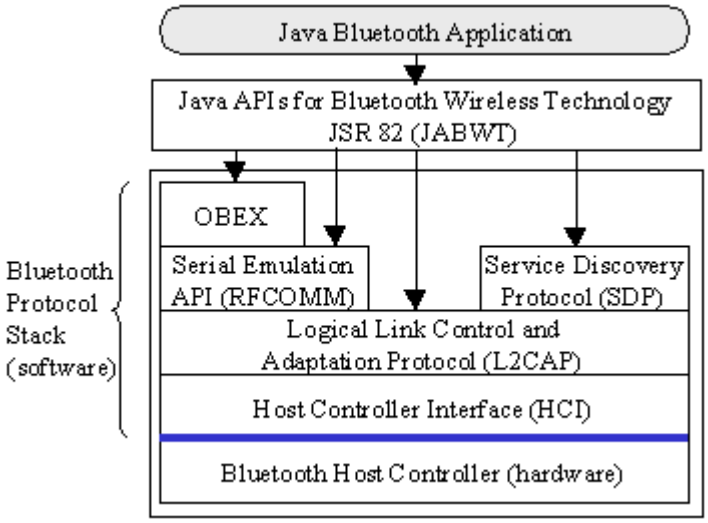

Figure 2. JSR 82 and Bluetooth Stack

L2CAP segments data into packets for transmission, and re-assembles received packets into larger messages.

RFCOMM provides functionality similar to a standard serial communications port, which is utilized as a stream connection at the Java level. I employ RFCOMM as the communications layer in this chapter's echoing client/server example.

OBEX (the Object Exchange protocol) provides a means for exchanging objects between objects (such as images and files), and is built on top of RFCOMM

SDP allows a server to register its services with the device, and is also employed by clients looking for devices and services. The Java Bluetooth API hides SDP behind a discovery API supported by a DiscoveryAgent class and DiscoveryListener interface.

Sources of more information on JSR 82 and Bluetooth are listed at the end of this chapter.

2. An Overview of the Application

Figure 3 shows the class diagram for the server-side of the application. Only the public and protected methods are shown. I've left out the superclasses and interfaces in order to simplify the diagrams.

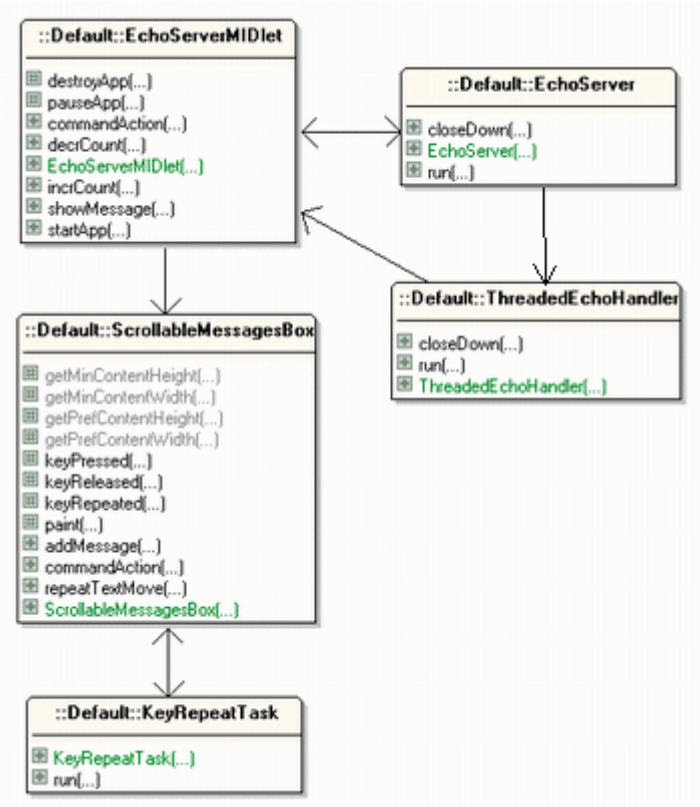

Figure 3. Class Diagrams for the Server.

EchoServerMIDlet is the top-level server-side MIDlet, and creates the GUI shown in the right-hand screen of Figure 1. It employs the ScrollableMessagesBox and KeyRepeatTask classes developed in "J2ME Chapter 1" to create a scrollable

messages area in the bottom part of the screen. EchoServer registers the server's Bluetooth echo 'service', and waits for client connections. Each connection is serviced by a new ThreadedEchoHandler thread.

The threaded nature of the application can be seen in Figure 4, which shows how the clients connect to the server.

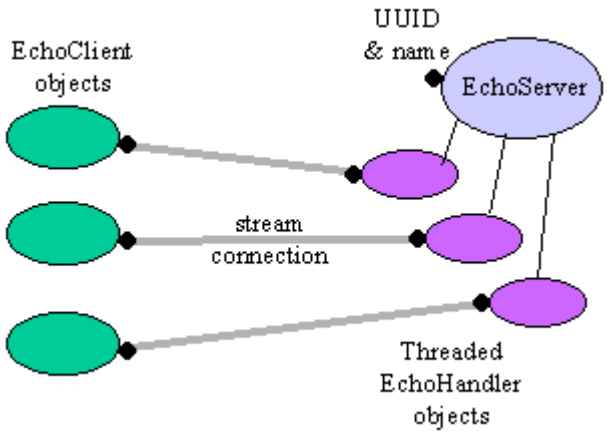

Figure 4. Client and Server Communication.

As I'll explain later, the echo server is identified by a Bluetooth UUID (a Universally Unique Identifier) and a service name. The stream connection between a client and handler is created using Bluetooth's RFCOMM protocol.

Figure 5 shows the classes used on the client-side of the application. As with Figure 3, the public and protected methods are shown, and the superclasses and interfaces are excluded.

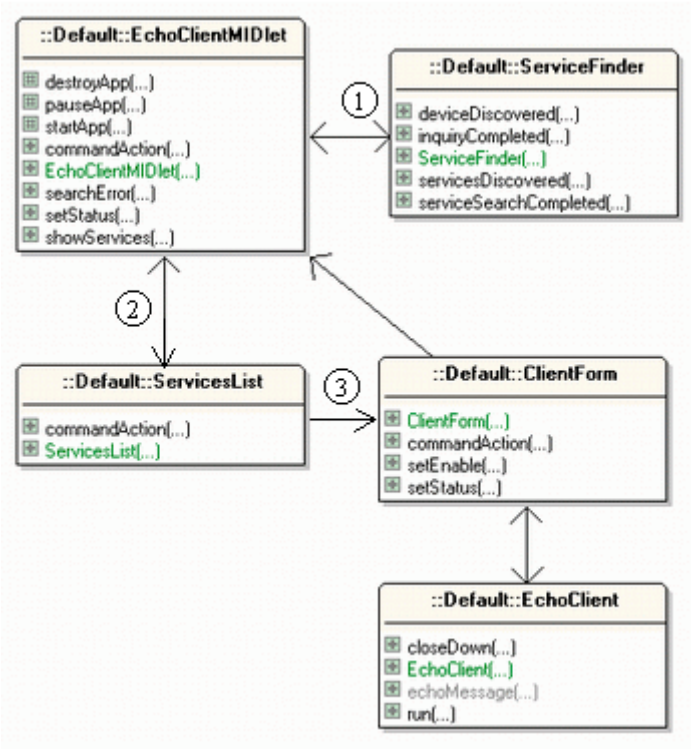

Figure 5. Class Diagrams for the Client.

EchoClientMIDlet is the top-level MIDlet for the client.

The numbered circles in Figure 5 indicate the order in which the classes are used: first ServiceFinder is employed to find suitable Bluetooth services, then the services are listed on-screen in ServicesList. The user chooses a service from the list, and ClientForm and EchoClient instances are created. ClientForm is the client-side GUI: it reads user input, and displays the server's replies (as shown in the left-hand screen of Figure 1). EchoClient handles the network communication with the server.

3. The Server MIDlet

EchoServerMIDlet starts the echo server, and acts as the server's GUI. It displays the current number of handlers, and a scrollable text box of server messages (see Figure 1). The text box is a ScrollableMessagesBox instance, called scroller.

The echo server is created in startApp() when the MIDlet starts executing:

```
// globals
private EchoServer echoServer;  // the top-level echo server


protected void startApp()
{
  display = Display.getDisplay(this);
```



```
  display.setCurrent(form);
  echoServer = new EchoServer(this);   // create the echo server
  echoServer.start();   // start it listening for connections
}
```

The MIDlet has several public methods which allow the echo server (and its threaded handlers) to adjust the handlers count field and add messages to the scrolling text box:

```
// globals
private StringItem numHandlersSI, messageSI;
private ScrollableMessagesBox scroller;
private int numHandlers = 0;   // number of threaded echo handlers


synchronized public void incrCount()
// called when a new handler is created
{ numHandlers++;
  numHandlersSI.setText("" + numHandlers);
}

synchronized public void decrCount()
// called when a handler is about to terminate
{ numHandlers--;
  numHandlersSI.setText("" + numHandlers);
}

synchronized public void showMessage(String msg)
// used to show the message received by a handler
{ scroller.addMessage(msg); }
```

The methods are synchronized since it's possible that two or more handlers could call them simultaneously.

Selecting EchoServerMIDlet's "Exit" command triggers a shut-down of the server and its handlers via the closeDown() method in EchoServer:

```
public void commandAction(Command c, Displayable d)
{ if (c == exitCmd) {
    echoServer.closeDown();  // close down server and its handlers
    destroyApp(true);
  }
}
```

4. The Echo Server

EchoServer is a threaded RFCOMM-based echo service, identified by a UUID (a unique Bluetooth ID) and service name ("echoserver").

The waiting for client connections is done in a thread so that it doesn't cause the top-level MIDlet to block. When a client does connect, a ThreadedEchoHandler thread is spawned to deal with it.

4.1. Service Initialization

The service is initialized in EchoServer():

```
// globals
// UUID and name of the echo service
private static final String UUID_STRING =
       "11111111111111111111111111111111";
       // 32 hex digits which will become a 128 bit ID
private static final String SERVICE_NAME = "echoserver";

private EchoServerMIDlet ecm;
private StreamConnectionNotifier server;

private Vector handlers;  // stores the ThreadedEchoHandler objects


public EchoServer(EchoServerMIDlet ecm)
{
  this.ecm = ecm;
  handlers = new Vector();
  try {  // make the server's device discoverable
    LocalDevice local = LocalDevice.getLocalDevice();
    local.setDiscoverable(DiscoveryAgent.GIAC);
  }
  catch (BluetoothStateException e) {
    System.out.println(e);
    return;
  }

  /* Create a RFCOMM connection notifier for the server, with
     the given UUID and name. This also creates a service
     record. */
  try {
    server = (StreamConnectionNotifier) Connector.open(
        "btspp://localhost:" + UUID_STRING +
                            ";name=" + SERVICE_NAME);
  }
  catch (IOException e) {
    System.out.println(e);
    return;
  }
}  // end of EchoServer()
```

The server's device must be discoverable by a client:

```
  LocalDevice local = LocalDevice.getLocalDevice();
  local.setDiscoverable(DiscoveryAgent.GIAC);
```

The DiscoveryAgent.GIAC (General/Unlimited Inquiry Access Code) constant means that all remote devices (i.e. all the clients) will be able to find the device. There's also a DiscoveryAgent.LIAC constant which limits the device's visibility.

The RFCOMM stream connection offered by the server requires a suitably formatted URL. The basic format is:

```
  btspp://<hostname>:<UUID>;<parameters>
```

I use `localhost` as the hostname, but any Bluetooth address can be employed.

The UUID field is a unique 128-bit identifier representing the service; I utilize a 32 digit hexadecimal string (each hex digit uses 4 bits).

The URL's parameters are "<name>=<value>" pairs, separated by semicolons. Typical <name> values are "name" for the service name (used here), and security parameters such as "authenticate", "authorize", and "encrypt". In WTK 2.2., the value of the "name" string must be in lowercase or a client won't recognize the service name at discovery time.

The other main URL type is for L2CAP connections, where messages are sent as packets, in a similar style to UDP datagrams.

The creation of the StreamConnectionNotifier instance, server, by the Connector.open () call also generates an implicit service record. The record is a description of the Bluetooth service as a set of (id, value) attributes. It can be accessed by calling LocalDevice.getRecord():

```
ServiceRecord record = local.getRecord(server);
```

The ServiceRecord class offers get/set methods for accessing and changing a record's attributes.

4.2. Waiting for a Client

EchoServer waits for client connections in a separate thread (EchoServer implements Thread) so that the enclosing MIDlet isn't blocked.

```
// global
private boolean isRunning = false;


public void run()
// Wait for client connections, creating a handler for each one
{
  isRunning = true;
  try {
    while (isRunning) {
      StreamConnection conn = server.acceptAndOpen();
      ThreadedEchoHandler hand = new ThreadedEchoHandler(conn, ecm);
      handlers.addElement(hand);
      hand.start();
    }
  }
  catch (IOException e)
  {  System.out.println(e);  }
}  // run()
```

The call to acceptAndOpen() makes the server block until a client connection arrives, and also adds the server's service record to the device's Service Discovery Database (SDDB). When a client carries out device and service discovery it contacts the SDDBs of the devices that it's investigating.

When a client connection is made, acceptAndOpen() returns a MIDP StreamConnection object, which is passed to a ThreadedEchoHandler instance so it can deal with the client communication.

4.3. Closing Down

The server and handlers are terminated by EchoServerMIDlet calling EchoServer's closeDown() method.

```
public void closeDown()
{
  System.out.println("Close down server");
  if (isRunning) {
    isRunning = false;
    try {
      server.close();
         // close connection, and remove service record from SDDB
    }
    catch (IOException e)
    {  System.out.println(e);  }

    // close down the handlers
    ThreadedEchoHandler hand;
    for (int i=0; i < handlers.size(); i++) {
       hand = (ThreadedEchoHandler) handlers.elementAt(i);
       hand.closeDown();
    }
    handlers.removeAllElements();
  }
}  // end of closeDown();
```

The closing of the StreamConnectionNotifier also instructs the SDDB to delete the server's service record.

The handlers are terminated by calling their closeDown() method, which close the stream connections to the clients.

5. The Threaded Echo Handler

A handler begins by extracting information about the client from the StreamConnection instance.

```
// globals
private EchoServerMIDlet ecm;  // top-level MIDlet
private StreamConnection conn; // client connection
private String clientName;


public ThreadedEchoHandler(StreamConnection conn,
                           EchoServerMIDlet ecm)
{
  this.conn = conn;
  this.ecm = ecm;

  System.out.println("Client Handler spawned");
  // store the name of the connected client
  try {
    RemoteDevice rd = RemoteDevice.getRemoteDevice(conn);
```



```
      clientName = getDeviceName(rd);
      System.out.println("Client name: " + clientName);
    }
    catch(Exception e) {}
} // end of ThreadedEchoHandler()


private String getDeviceName(RemoteDevice dev)
/* Return the 'friendly' name of the device being examined,
   or "Device ??" */
{
  String devName;
  try {
    devName = dev.getFriendlyName(false);
  }
  catch (IOException e)
  { devName = "Device ??";  }
  return devName;
}
```

The information about a remote device (the client in this case) is stored in a RemoteDevice instance. RemoteDevice includes methods for finding the client's Bluetooth address, its 'friendly' device name, details about its security settings, and for verifying those settings.

The handler retrieves the client's device name by calling RemoteDevice.getFriendlyName():

```
    devName = dev.getFriendlyName(false);
```

The `false` argument stops the handler from obtaining the name by contacting the client via a new connection. Instead, information present in the RemoteDevice object (dev) is utilized.

If `true` is used, it's possible for an IOException to be raised on certain phones, because the requested connection may take the total number of connections past the maximum allowed for the device.

5.1. How Many Connections?

A device's permitted maximum number of concurrent Bluetooth connections can be obtained by calling LocalDevice.getProperty() with a "bluetooth.connected.devices.max" argument:

```
int maxConnections = Integer.parseInt(
  LocalDevice.getProperty("bluetooth.connected.devices.max"));
```

maxConnections can be as large as 7, but might only be 1. A value of 1 means that a client hosted on the device can only connect to a single server. Alternatively, if the device holds a server, then it can only have at most one client! This constraint would make the device fairly useless as a server.

In WTK 2.2., the emulator's value for "bluetooth.connected.devices.max" can be set in the "System Properties" tab of the "Bluetooth/OBEX" tab in the "Preferences" menu.

5.2. Connecting to the Client

ThreadedEchoHandler's run() method creates input and output streams, and starts processing the client's messages.

```
// globals
private InputStream in;
private OutputStream out;


public void run()
{
  ecm.incrCount();   // tell top-level MIDlet there's a new handler
  try {
    // Get I/O streams from the stream connection
    in = conn.openInputStream();
    out = conn.openOutputStream();
    processClient();
  }
  catch(Exception e)
  {  System.out.println(e);  }

  ecm.decrCount();  // remove this handler from the top-level count
}  // end of run()
```

An InputStream and OutputStream are extracted from the StreamConnection instance, and used in processClient(). The use of run() means that any I/O blocking will be in a separate thread from the server and its GUI.

It's possible to map a DataInputStream and DataOutputStream to the StreamConnection instance, so that basic Java data types (e.g. integers, floats, doubles, strings) can be read and written. I've used InputStream and OutputStream because their byte-based read() and write() methods can be easily utilized as 'building blocks' for implementing different forms of message processing (as shown below).

5.3. Talking to a Client

The processClient() method waits for a message to arrive from the client, converts it to uppercase, and sends it back. If the message is "bye$$", then the client wants to close the link.

```
private void processClient()
{
  isRunning = true;
  String line;
  while (isRunning) {
    if((line = readData()) == null)
      isRunning = false;
    else {  // there was some input
      if (line.trim().equals("bye$$"))
        isRunning = false;
      else {
        ecm.showMessage(clientName + ": " + line);
              // show in the GUI
        String upper = line.trim().toUpperCase();
```



```
        if (isRunning)
          sendMessage(upper);
      }
    }
  }
  System.out.println("Handler finished");
}  // end of processClient()
```

The messy details of reading a message are hidden inside readData(), which either returns the message as a string, or null if there's been a problem. A message is transmitted with sendMessage().

5.4. Reading a Message

When a client sends a message to the handler (e.g. "hello"), it is actually sent as a stream of bytes prefixed with its length (e.g. "5hello"). The number is encoded in a single byte, which puts an upper limit on the message's length of 255 characters.

Since a message always begins with its length, readData() can use that value to constrain the number of bytes it reads from the input stream.

```
private String readData()
{
  byte[] data = null;
  try {
    int len = in.read();    // get the message length
    if (len <= 0) {
      System.out.println("Message Length Error");
      return null;
    }

    data = new byte[len];
    len = 0;
    // read the message, perhaps requiring several read() calls
    while (len != data.length) {
      int ch = in.read(data, len, data.length - len);
      if (ch == -1) {
        System.out.println("Message Read Error");
        return null;
      }
      len += ch;
    }
  }
  catch (IOException e)
  {  System.out.println("readData(): " + e);
     return null;
  }

  return new String(data);  // convert byte[] to String
}   // end of readData()
```

InputStream.read() is called repeatedly until the necessary number of bytes have been obtained. The bytes are converted into a String, and returned; the message length is discarded.

5.5. Sending a Message

sendMessage() adds the message's length to the front of a message, and it is sent out as a sequence of bytes:

```
private boolean sendMessage(String msg)
{
  try {
    out.write(msg.length());
    out.write(msg.getBytes());
    return true;
  }
  catch (Exception e)
  {  System.out.println("sendMessage(): " + e);
     return false;
  }
}
```

5.6. Closing Down the Handler

The server terminates the handler by calling its closeDown() method. The input and output streams are closed first, then the underlying StreamConnection.

```
public void closeDown()
{
  System.out.println("Close down echo handler");
  isRunning = false;
  try {
    if (conn != null) {
      in.close();
      out.close();
      conn.close();
    }
  }
  catch (IOException e)
  {  System.out.println(e);  }
}
```

isRunning is set to false, which will cause the I/O loop in processClient() to finish.

7. The Client MIDlet

EchoClientMIDlet's main task is to create an instance of ServiceFinder to carry out devices and services searches.

```
// globals
private static final String UUID_STRING =
                  "11111111111111111111111111111111";
private static final String SERVICE_NAME = "echoserver";

private ServiceFinder serviceFinder;

// in EchoClientMIDlet's constructor
serviceFinder = new ServiceFinder(this, UUID_STRING, SERVICE_NAME);
```

The required service will be identified by its UUID and service name, so they're passed to serviceFinder.

The searches may take many seconds, so EchoClientMIDlet displays a continuously running gauge (a waving penguin), as shown in Figure 6.

Figure 6. EchoClientMIDlet's Screen.

EchoClientMIDlet contains public methods for updating the status string and the gauge shown in Figure 6; these methods are mainly utilized by ServiceFinder to inform the user of the progress of its searches.

When ServiceFinder finishes, it returns control to the MIDlet, which displays the matching services (stored in searchTable) in a list managed by ServicesList.

```
public void showServices(Hashtable searchTable)
// called by ServiceFinder
{ ServicesList sl = new ServicesList(searchTable, this, display);
  display.setCurrent(sl);
}
```


8. Finding a Device, Finding a Service

The ServiceFinder class encapsulates the most complex aspects of writing a Bluetooth application – having a client search for devices, and then search each of those devices for relevant services. Section 8.1 explains devices search, while section 8.2 considers services search.

8.1. Devices Discovery

Figure 7 shows the stages involved in devices discovery:

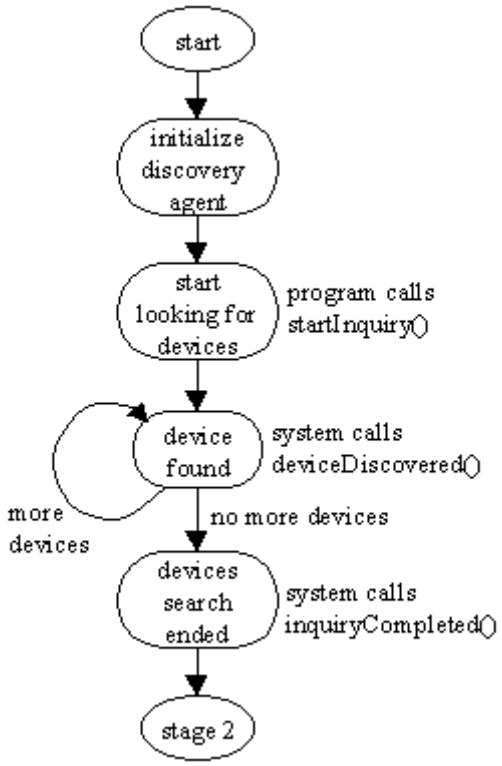

Figure 7. Device Discovery State Diagram.

The state diagram lists three methods that play important roles in the devices search: DiscoveryAgent.startInquiry(), DiscoveryListener.deviceDiscovered(), and DiscoveryListener.inquiryCompleted(). They will be explained in detail below.

Figure 7 ends in the "stage 2" state, which leads into the services search explained in section 8.2 and illustrated by Figure 8.

ServiceFinder begins by creating a discovery agent, then starts the devices search by calling DiscoveryAgent.startInquiry():

```
// globals
private EchoClientMIDlet ecm;
private String UUIDStr;           // UUID of the desired service
private String srchServiceName;   // name of the desired service

private DiscoveryAgent agent;

// stores the remote devices found during the device search
```



```
private Vector deviceList;


public ServiceFinder(EchoClientMIDlet ecm, String uuid, String nm)
{
  this.ecm = ecm;
  UUIDStr = uuid;
  srchServiceName = nm;

  try {
    // get the discovery agent, by asking the local device
    LocalDevice local = LocalDevice.getLocalDevice();
    agent = local.getDiscoveryAgent();

    // initialize device search data structure
    deviceList = new Vector();

    // start the searches: devices first, services later
    ecm.setStatus("Searching for Devices...");
    agent.startInquiry(DiscoveryAgent.GIAC, this);   // non-blocking
  }
  catch (Exception e) {
    System.out.println(e);
    ecm.searchError("Search Error");
  }
} // end of ServiceFinder()
```

The deviceList vector is used during the search to hold devices information in the form of RemoteDevice objects.

DiscoveryAgent.startInquiry() is non-blocking, so the system communicates its progress by calling DiscoveryListener.deviceDiscovered(), and DiscoveryListener.inquiryCompleted().

ServiceFinder implements the DiscoveryListener interface, and a reference to it (`this`) is passed to startInquiry() as its second argument. This means that ServiceFinder's deviceDiscovered() and inquiryCompleted() methods will be called during the search.

8.1.1. Finding a Device

deviceDiscovered() is called when a device is found. The method examines the device's details to decide if it should be added to deviceList.

```
public void deviceDiscovered(RemoteDevice dev, DeviceClass cod)
{
  System.out.println("Device Name: " + getDeviceName(dev));

  int majorDC = cod.getMajorDeviceClass();
  int minorDC = cod.getMinorDeviceClass();
  System.out.println("Major Device Class: " + majorDC +
                  "; Minor Device Class: " + minorDC);

  // restrict matching device to PC or Phone
  if ((majorDC == 0x0100) || (majorDC == 0x0200))
    deviceList.addElement(dev);
  else
    System.out.println("Device not PC or phone, so rejected");
```



```
} // end of deviceDiscovered()
```

It's a good idea to do as much 'filtering' of devices as possible at this stage, to keep deviceList small. This will speed up the subsequent services search since fewer devices will need to be contacted.

deviceDiscovered()'s two arguments are RemoteDevice and DeviceClass objects.

A RemoteDevice instance maintains some useful filtering information, such as the remote device's name. That name is extracted with getDeviceName(), the same method as seen in ThreadedEchoHandler. However, the primary purpose of RemoteDevice is for contacting the remote device.

Most device filtering is done by examining the device's DeviceClass object, which represents a Bluetooth Class of Device (CoD) record. A CoD record contains the device's major class ID, minor class ID, and service classes type. DeviceClass offers getMajorDeviceClass(), getMinorDeviceClass(), and getServiceClasses() for accessing these details.

A complete list of the CoD classifications can be found at Bluetooth.org, at https://www.bluetooth.org/foundry/assignnumb/document/baseband.

The major device classification is a broad functional category, such as 'computer', 'phone', or 'imaging device'. My version of deviceDiscovered() only stores a device if it is a computer or phone.

A device may belong to several minor device classifications, and so it's necessary to use a bit mask to check if the device is in a particular category. For example:

```
int majorDC = cod.getMajorDeviceClass();
int minorDC = cod.getMinorDeviceClass();

// check if the device is a printer
if (majorDC == 0x600)   // device is an imaging device
  if (minorDC & 0x80)   // device is a printer
    System.out.println("Device is a printer");
```

A device may offer multiple services, such as networking, rendering, capture, and audio, so bit masks are required again:

```
int serviceClass = cod.getServiceClasses();
if (serviceClass & 0x20000) != 0)
  System.out.println("Device supports networking");
if (serviceClass & 0x40000) != 0)
  System.out.println("Device supports rendering");
```

8.1.2. The End of the Devices Search

When no more devices can be found, the system calls DiscoveryListener.inquiryCompleted().

```
public void inquiryCompleted(int inqType)
// device search has finished; start the services search
{
```



```
  showInquiryCode(inqType);
  System.out.println("No. of Matching Devices: " +
                                    deviceList.size());
  // start the services search
  ecm.setStatus("Searching for Services...");
  searchForServices(deviceList, UUIDStr);
} // end of inquiryCompleted()


private void showInquiryCode(int inqCode)
{
  if(inqCode == INQUIRY_COMPLETED)
    System.out.println("Device Search Completed");
  else if(inqCode == INQUIRY_TERMINATED)
    System.out.println("Device Search Terminated");
  else if(inqCode == INQUIRY_ERROR)
    System.out.println("Device Search Error");
  else
    System.out.println("Unknown Device Search Status: " + inqCode);
}  // end of showResponseCode()
```

inquiryCompleted() is passed an inquiry completion integer, indicating how the search terminated. Normal termination is represented by DiscoveryListener.INQUIRY_COMPLETED.

The main task of inquiryCompleted() is to initiate the services search by calling searchForServices(). This coding approach means that the services search doesn't start until the devices search is completed.

Alternatively, it is possible to start the services search in deviceDiscovered(), which will trigger simultaneous devices and services searching. However, many phones don't support this functionality.

8.1.3. Making the Devices Search Faster

The devices search may take up to 10 seconds to complete, which is too long for many applications. One way of reducing this time is to cut short the search as soon as a suitable device has been found.

The search can be terminated by calling cancelInquiry():

```
  agent.cancelInquiry(this);
```

`this` refers to the object implementing the DiscoveryListener interface, which is ServiceFinder in this example.

There will be no further calls to deviceDiscovered(), and the inquiryCompleted() argument will be DiscoveryListener.INQUIRY_TERMINATED.

Another possible speed optimization is to examine the cached and preknown RemoteDevice information for a suitable device before embarking on a devices search:

```
RemoteDevices[] cachedDevs =
  agent.retrieveDevices(DiscoveryAgent.CACHED);
RemoteDevices[] knownDevs =
```



```
agent.retrieveDevices(DiscoveryAgent.PREKNOWN);
```

Cached devices are ones discovered in previous searches, while preknown devices are those that are frequently contacted. Many phones don't support preknown devices.

Using cached information may avoid the need for a devices search, but unfortunately the CoD (class of device) records for the devices are not cached. This means that it's impossible to quickly determine which cached devices are relevant to the present search. Also, there's no indication of whether the cache is current.

8.2. Services Search

Figure 8 shows the main stages in a services search:

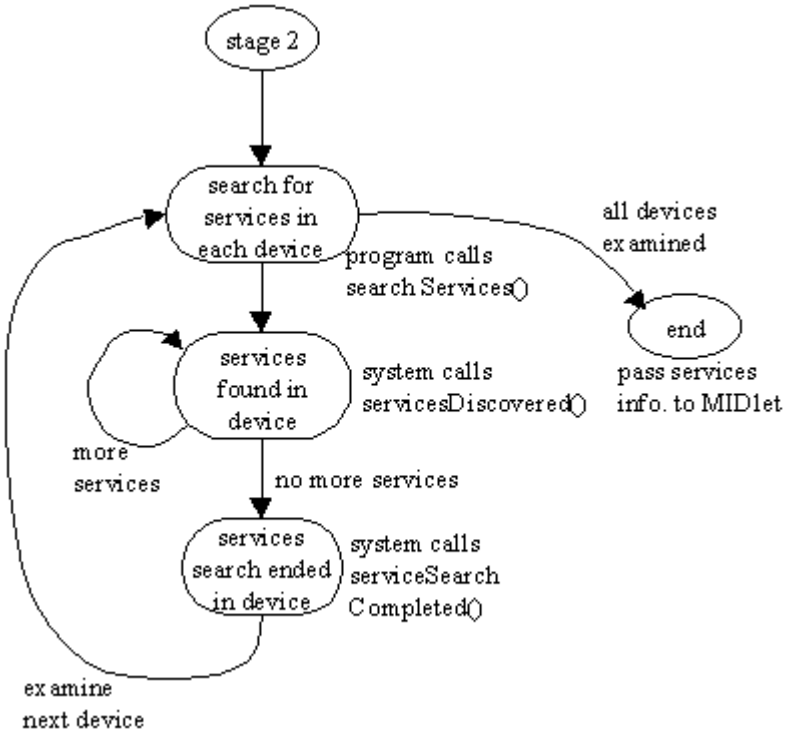

Figure 8. Services Search State Diagram.

Figure 8 starts where Figure 7 finishes. In the code, this corresponds to inquiryCompleted() calling searchForServices().

The state diagram consists of two loops: the outer loop cycles through each device found during the devices search, while the inner loop checks each service offered by a particular device.

The figure mentions three methods that play important roles in services search: DiscoveryAgent.searchServices(), DiscoveryListener.servicesDiscovered(), and DiscoveryListener.serviceSearchCompleted(). They will be explained in detail below.

The outer loop of the state diagram is implemented in searchForServices(), which iterates through the deviceList vector:

```
// globals
private String UUIDStr;           // the UUID of the desired service
private String srchServiceName;   // the name of the desired service

/* table of matching services, stored as pairs of the form
   {device name, serviceRecord} */
private Hashtable serviceTable;


private void searchForServices(Vector deviceList, String UUIDStr)
/* Carry out service searches for all the matching devices, looking
   for the RFCOMM service with UUID == UUIDStr. Also check the
   service name.  */
{
  UUID[] uuids = new UUID[2];   // holds UUIDs used in the search

  /* Add the UUID for RFCOMM to make sure that the matching service
     support RFCOMM. */
  uuids[0] = new UUID(0x0003);

  // add the UUID for the service we're looking for
  uuids[1] = new UUID(UUIDStr, false);

  /* we want the search to retrieve the service name attribute,
     so we can check it against the service name we're looking for */
  int[] attrSet = {0x0100};

  // initialize service search data structure
  serviceTable = new Hashtable();

  // carry out a service search for each of the devices
  RemoteDevice dev;
  for (int i = 0; i < deviceList.size(); i++) {
    dev = (RemoteDevice) deviceList.elementAt(i);
    searchForService(dev, attrSet, uuids);
  }

  // tell the top-level MIDlet the result of the searches
  if (serviceTable.size() > 0)
    ecm.showServices(serviceTable);
  else
    ecm.searchError("No Matching Services Found");
} // end of searchForServices()
```

Before it starts examining the devices, searchForServices() sets up the search criteria, which is to match on a RFCOMM service with the correct UUID, and retrieve the service's name. Hex values for the many possible UUIDs and attributes are listed at https://www.bluetooth.org/foundry/assignnumb/document/service_discovery.

searchForServices() also initializes the principal search data structure, a Hashtable called serviceTable. serviceTable will hold matching services, stored as {device name, serviceRecord} pairs.

When the for-loop finishes, every device will have been searched, and all the matching services will have been added to serviceTable. The Hashtable is then passed to the top-level MIDlet.

searchForService() calls DiscoveryAgent.searchServices() to initiate a search of a given device, and uses waitForSearchEnd() to wait for the search to finish.

```
private void searchForService(RemoteDevice dev,
                              int[] attrSet, UUID[] uuids)
// search for matching services on this device (dev)
{
  System.out.println("Searching device: " + getDeviceName(dev));
  try {
    int trans = agent.searchServices(attrSet, uuids, dev, this);
      // non-blocking search
    waitForSearchEnd(trans);
  }
  catch (BluetoothStateException e) {
    System.out.println(e);
  }
}  // end of searchForService()
```

The call to searchServices() includes the attributes and UUIDs arrays which constrain the search. The fourth argument is a reference to the DiscoveryListener (this object) whose servicesDiscovered() and serviceSearchCompleted() methods will be called.

searchServices() returns a transaction ID for the search.

waitForSearchEnd() uses wait() to stop ServiceFinder until the searchDone boolean is set to true by serviceSearchCompleted() (as described below). This signals that the current device has been completly searched.

```
// global
private boolean searchDone;


private void waitForSearchEnd(int trans)
// wait for the current service search to finish
{
  System.out.println("Waiting for trans ID " + trans + "...");
  searchDone = false;
  while (!searchDone) {
    synchronized (this) {
      try {
        this.wait();
      }
      catch (Exception e) {}
    }
  }
  System.out.println("Done");
}  // end of waitForSearchEnd()
```


When waitForSearchEnd() resumes, searchForService() can return, and searchForServices() can consider the next device. This coding approach means that only one device will be examined at a time.

8.2.1. Concurrent Services Searches

Since DiscoveryAgent.searchServices() is non-blocking, it's possible to carry out concurrent services searches by commenting out the call to waitForSearchEnd() in searchForService(). searchForServices() will call searchForService() multiple times in rapid succession, generating a concurrent services search for each device.

Unfortunately, most phones don't support concurrent services searches, so the concurrent searchServices() calls will either block or fail.

A device's support for concurrent services searches can be discovered by examining its "bluetooth.sd.trans.max" property:

```
int maxConServiceSearches = Integer.parseInt(
          LocalDevice.getProperty("bluetooth.sd.trans.max"));
```

The value will be 1 on most phones. In the WTK emulator, the number can be adjusted via the "System Properties" tab of the "Bluetooth/OBEX" tab in the "Preferences" menu.

Another reason for avoiding concurrent searches is that they make it more tricky to implement early search termination. By contrast, the termination of a sequential search is straightforward, as detailed in section "8.2.4. Making the Services Search Faster".

8.2.2. Finding Services in a Device

The system calls DiscoveryListener.servicesDiscovered() whenever matching services are discovered in the device. Depending on the device, servicesDiscovered() may be called once with all the matching services, or perhaps several times with a single service each time. This behaviour corresponds to the inner loop in Figure 8.

```
public void servicesDiscovered(int transID,
                           ServiceRecord[] servRecords)
{
  for (int i=0; i < servRecords.length; i++) {
    if (servRecords[i] != null) {
      // get the service record's name
      DataElement servNameElem =
                 servRecords[i].getAttributeValue(0x0100);
      String servName = (String)servNameElem.getValue();
      System.out.println("Name of Discovered Service: " + servName);

      if (servName.equals(srchServiceName)) {  // check the name
        RemoteDevice dev = servRecords[i].getHostDevice();
        serviceTable.put( getDeviceName(dev), servRecords[i]);
           // add to table
      }
    }
  }
```



```
} // end of servicesDiscovered()
```

The exact meaning of a 'matching' service depends on the parameters supplied to DiscoveryAgent.searchServices(). My code specifies that the service UUID be equal to UUIDStr ("11111111111111111111111111111111") and employ the RFCOMM protocol (a stream connection). Also, the retrieved service record should include the service name.

servicesDiscovered() extracts the service name from the service record, and tests it against srchServiceName ("echoserver"). If the strings match, then the record is added to serviceTable, using the device name as the key.

8.2.3. Moving on to the Next Device

When the current device has been completely examined, DiscoveryAgent.serviceSearchCompleted() is called:

```
public void serviceSearchCompleted(int transID, int respCode)
{
  showResponseCode(transID, respCode);

  /* Wake up waitForSearchEnd() for this search, allowing the next
     services search to commence in searchForServices(). */
  searchDone = true;
  synchronized (this) {
    this.notifyAll();  // wake up
  }
} // end of serviceSearchCompleted()


private void showResponseCode(int transID, int respCode)
{
  System.out.print("Trans ID " + transID + ". ");

  if(respCode == SERVICE_SEARCH_ERROR)
    System.out.println("Service Search Error");
  else if(respCode == SERVICE_SEARCH_COMPLETED)
    System.out.println("Service Search Completed");
  else if(respCode == SERVICE_SEARCH_TERMINATED)
    System.out.println("Service Search Terminated");
  else if(respCode == SERVICE_SEARCH_NO_RECORDS)
    System.out.println("Service Search: No Records found");
  else if(respCode == SERVICE_SEARCH_DEVICE_NOT_REACHABLE)
    System.out.println("Service Search: Device Not Reachable");
  else
    System.out.println("Unknown Service Search Status: " + respCode);
}  // end of showResponseCode()
```

serviceSearchCompleted() reports the search termination status, then wakes up waitForSearchEnd(). When waitForSearchEnd() returns, searchForService() can finish, allowing searchForServices() to search the next device.

8.2.4. Making the Services Search Faster

Each services search of a device may take up to 4 seconds (this is in addition to the 10 seconds spent earlier on the devices search).

One of way of speeding up the services search stage is to stop it as soon as a single matching service is found. This makes particular sense for this application since the client only needs to communicate with a single server.

The search is terminated early by jumping out of the devices for-loop in searchForServices():

```
// in searchForServices()
boolean searchesTerminated = false;
RemoteDevice dev;

for (int i = 0; i < deviceList.size(); i++) {
  dev = (RemoteDevice) deviceList.elementAt(i);
  searchForService(dev, attrSet, uuids);
  if (searchesTerminated)
    break;
}
```

The searchesTerminated boolean can be set to true in servicesDiscovered() when a suitable service record is found.

This approach stops the search of any further devices after the current one. To stop the current search requires a call to DiscoveryAgent.cancelServiceSearch() in servicesDiscovered():

```
  agent.cancelServiceSearch(transID);
```

transID should be the transaction ID of the current search.

9. Choosing from the Services List

The result of executing ServiceFinder is a Hashtable of matching services. Those services are passed to EchoClientMIDlet via a call to its showServices() method:

```
// in EchoClientMIDlet
public void showServices(Hashtable searchTable)
// called by ServiceFinder
{ ServicesList sl = new ServicesList(searchTable, this, display);
  display.setCurrent(sl);
}
```


searchTable is handed onto ServicesList, a subclass of MIDP's List. The device names for the services are displayed on-screen, and the user selects one. Figure 9 shows a typical ServicesList screen:

Figure 9. A ServicesList Screen.

The selection of a device name triggers a call to commandAction():

```
public void commandAction(Command c, Displayable d)
{
  if ((c == startCmd) || (c == List.SELECT_COMMAND)) {
    // list item selected
    int index = getSelectedIndex();
    String devName = deviceNames[index];   // get the device name
    System.out.println("Selected echo server: " + devName);

    ServiceRecord sr = (ServiceRecord) servicesTable.get(devName);
            // get the corresponding service record for the name

    ClientForm cf = new ClientForm(sr, ecm, display);
    display.setCurrent(cf);
        /* Display the client form, which carries out the
           interaction with the service. */
  }
  else if (c == exitCmd)
    ecm.destroyApp(true);
}  // end of commandAction()
```

The selected device name is used to lookup the matching service record in the Hashtable. This record is passed to an instance of ClientForm, which acts as the GUI for the client's interactions with the echo server.

10. The Client GUI

ClientForm is a form-based GUI for the EchoClient object which manages the sending and receiving of messages with the echo server.

The form consists of a text field (messageTF) for entering a message, and two text lines, one showing the last response from the server, the other the status of the server link. The GUI is shown in Figure 10.

Figure 10. The ClientForm GUI.

The networking part of the client, the EchoClient object, is created and started in ClientForm's constructor:

```
// global
private EchoClient echoClient;
  // handles the Bluetooth communication with the server

// in the constructor
echoClient = new EchoClient(sr, this);
echoClient.start();   // connect to the server
```

sr is the service record, and the ClientForm reference (this) allows EchoClient to update the response and status strings in the form.

A message in the messageTF text field is sent by the user clicking on the "Send" command. This is processed in commandAction():

```
public void commandAction(Command c, Displayable d)
{
  if (c == exitCmd) {
    echoClient.closeDown();
    ecm.destroyApp(true);
  }
  else if (c == sendCmd) {
    String resp = echoClient.echoMessage( messageTF.getString() );
    responseSI.setText(resp);  // show the response
  }
}  // end of commandAction()
```

The message is sent to the server via EchoClient's echoMessage() method. That method waits for an answer from the server, which is written into the response string.

If EchoClient detects an error when communicating with the server, it notifies the client by setting the status string, and disabling any further input from the text field by using ClientForm's setStatus() and setEnable() methods:


```
public void setStatus(String msg)
// report the status of the server connection
{  statusSI.setText(msg);  }


public void setEnable(boolean isEnabled)
// enable/disable the text field
{
  if (isEnabled) {
    messageTF.setConstraints(TextField.ANY);
    addCommand(sendCmd);
  }
  else {   // disable the text field
    messageTF.setConstraints(TextField.ANY | TextField.UNEDITABLE);
    removeCommand(sendCmd);
  }
}
```

11. Communicating with the Server

EchoClient is passed the server's service record, which it uses to create a RFCOMM stream connection. The connection is mapped to an OutputStream and InputStream for sending and receiving messages.

The opening of the connection is carried out in a thread since Connector.open() may block for a long period, and that shouldn't affect the ClientForm GUI.

```
// globals
private ServiceRecord servRecord;
private ClientForm clientForm;

private StreamConnection conn;  // for the server
private InputStream in;     // stream from server
private OutputStream out;   // stream to server

private boolean isClosed = true;
  // is the connection to the server closed?


public void run()
{
  // get a URL for the service
  String servURL = servRecord.getConnectionURL(
                     ServiceRecord.NOAUTHENTICATE_NOENCRYPT, false);

  if (servURL != null) {
    clientForm.setStatus("Found Echo Server URL");
    try {
      // connect to the server, and extract IO streams
      conn = (StreamConnection) Connector.open(servURL);
      out = conn.openOutputStream();
      in = conn.openInputStream();

      clientForm.setStatus("Connected to Echo Server");
      clientForm.setEnable(true);   // communication allowed
      isClosed = false;   // i.e. the connection is open
    }
```



```
    catch (Exception ex)
    {  System.out.println(ex);
       clientForm.setStatus("Connection Failed");
    }
  }
  else
    clientForm.setStatus("No Service Found");
}  // end of run()
```

The first argument of the call to ServiceRecord.getConnectionURL() specifies the level of security for the connection; I've opted for no security. The second argument relates to the master-slave communications protocol at the Bluetooth level, and should usually be set to false.

Once the connection URL has been extracted from the service record, a stream connection is obtained with Connector.open(), and an InputStream and OutputStream are layered on top of it.

I use InputStream and OutputStream for the same reason as in the server – additional message passing functionality is easily implemented with read() and write().

11.1. Sending a Message

A message is sent out when ClientForm calls echoMessage(). The method waits for an response then returns it to ClientForm.

```
public String echoMessage(String msg)
{
  if (isClosed) {
    disableClient("No Connection to Server");
    return null;
  }

  if ((msg == null) || (msg.trim().equals("")))
    return "Empty input message";
  else {
    if (sendMessage(msg)) {    // message sent ok
      String response = readData();  // wait for response
      if (response == null) {
        disableClient("Server Terminated Link");
        return null;
      }
      else  // there was a response
        return response;
    }
    else {   // unable to send message
      disableClient("Connection Lost");
      return null;
    }
  }
}  // end of echoMessage()
```

If there's a problem, the client is notified using disableClient(), and null is returned.

The readData() and sendMessage() methods employed in echoMessage() are the same as the ones used in ThreadedEchoHandler.

11.2. Closing Down

closeDown() sends a "bye$$" message to the server, then closes the connection.

```
public void closeDown()
{
  if (!isClosed) {
    sendMessage("bye$$");  // tell server that client is leaving
    try {
      if (conn != null) {
        in.close();
        out.close();
        conn.close();
      }
    }
    catch (IOException e)
    {  System.out.println(e);  }
    isClosed = true;
  }
}  // end of closeDown();
```

12. More Information on J2ME/Java and Bluetooth

All the current J2ME emulators support Bluetooth. For example:

- Sun's J2ME Wireless Toolkit 2.2 (WTK 2.2)
  http://java.sun.com/products/sjwtoolkit/download-2_2.html
- Nokia Developer's Suite 2.2 for J2ME
  http://www.forum.nokia.com/main/0,6566,034-2,00.html
- Sony Ericsson SDK 2.2.2 for J2ME
  http://developer.sonyericsson.com

One reason for downloading *all* of these emulators is to obtain the different Bluetooth examples they contain.

BlueCove is a free implementation of the Bluetooth stack for **J2SE** on Windows XP, but it assumes the presence of Bluetooth hardware in the PC (http://sourceforge.net/projects/bluecove). The Benhui.net site has two articles on how to link BlueCove on a PC with Bluetooth-enabled phones (http://www.benhui.net/modules.php?name=Bluetooth).

There are several commercial development kits for J2ME/Java and Bluetooth. A good list can be found at http://www.javabluetooth.com/development_kits.html.

JSR 82 information can be downloaded from http://www.jcp.org/aboutJava/communityprocess/final/jsr082/. The zip file includes API details in HTML format, source code, and the JSR 82 specification.

General Bluetooth information can be found at https://www.bluetooth.org/ and http://www.bluetooth.com.

One of the best J2ME+Bluetooth sites is Benhui.net (http://benhui.net/modules.php?name=Bluetooth). There's an active forum, links to FAQs and development tools, code examples (including a large chat application), and a Bluetooth device database.

Another good site is http://www.javabluetooth.com/, which includes a Bluetooth introduction, information on kits and devices, and a FAQ.

There are many forums and discussion groups related to Bluetooth and J2ME, including:

- Nokia Developers forum, http://forum.nokia.com
- Sony Ericsson Developers forum, http://developer.sonyericsson.com
- J2ME.org forum, http://www.j2me.org
- Mobile Game Developer forums
  http://www.mobilegd.com/modules.php?name=Forums
- The Java Games forum for J2ME
  http://192.18.37.44/forums/index.php?board=17.0

Many of these sites also include articles and Bluetooth development tools.

For beginners, the following reports may be helpful:

- *Using the Java APIs for Bluetooth Wireless Technology*, Parts 1 and 2
  C. Enrique Ortiz, December 2004, February 2005
  http://developers.sun.com/techtopics/mobility/apis/articles/bluetoothintro/
  and http://developers.sun.com/techtopics/mobility/apis/articles/bluetoothcore/
- *Introduction to Bluetooth and J2ME*, Parts 1 and 2
  Jason Lam, December 2004
  http://wirelesspronews.com/wirelesspronews-14-20041213IntroductiontoBluetoothandJ2ME.html
  and http://www.jasonlam604.com/articles_introduction_to_bluetooth_and_j2me_part2.php

The Bluetooth Learning Guide at SearchMobileComputing.com is a good source of general articles and links on Bluetooth (http://searchmobilecomputing.techtarget.com/originalContent/0,289142,sid40_gci943232,00.html).

If you prefer books, then the following all have a chapter on Bluetooth and J2ME:

- *Programming Java 2 Micro Edition on Symbian OS: A Developer's Guide to MIDP 2.0*
  Martin de Jode, John Wiley, July 2004
  http://www.symbian.com/books/j2me/j2me_info.html
- *Nokia Series: Developing Scalable Series 40 Applications – A Guide for Java Developers*
  Michael Juntao Yuan, Addison Wesley, December 2004
  http://www.enterprisej2me.com/pages/series40/book.php

  This book contains two chapters on Bluetooth.
- *Beginning J2ME: From Novice to Professional*, 3rd Edition
  Sing Li and Jonathan Knudsen, Apress, May 2005
  http://www.apress.com/book/bookDisplay.html?bID=426

The URLs point to the book's source code, and other resources.

There are two slightly older books which emphasize Java and Bluetooth:

- *Bluetooth for Java*
  Bruce Hopkins and Ranjith Antony, Apress, March 2003
  http://www.apress.com/book/bookDisplay.html?bID=139 and
  http://www.javabluetooth.com/
- *Bluetooth Application Programming with the Java APIs*
  C Bala Kumar, Paul Kline, Tim Thompson
  Morgan Kaufmann, September 2003
  http://books.elsevier.com

