

Chapter B4. An Echoing Client/Server Application Using BlueCove

In many ways, this chapter is similar to chapter B1, in that it's about an echo client/server application using Bluetooth. It deserves it's own chapter because I've rewritten the code to use the BlueCove implementation of the Java Bluetooth API (<http://bluecove.org/>), aimed at JavaSE. This means that the midlet GUI had to be changed to Swing – a fairly easy task. A more important, and difficult, job was revamping the device and service discovery code. The standard approach, described in chapter B1, isn't robust enough to deal with the radio interference in my test environment. I made several changes to deal with the very lengthy discovery and search times, and infrequent connection failures.

I could write at length about the hassles of finding the 'right' Bluetooth stack for Windows XP, but you may be pleased to read that I've moved my moaning and groaning to chapter B3, "Bluetooth Programming Problems" (<http://fivedots.coe.psu.ac.th/~ad/jg/blue3/>). In essence, when the Bluetooth communication protocol is RFCOMM (as here), then you may as well stick with the Microsoft stack – it's a lot less hassle than trying to replace it with a more fully-featured implementation such as the Widcomm/Broadcom stack. I know this because the server in this chapter is running BlueCove on top of a Widcomm stack, while my test clients use BlueCove on Microsoft's own stack.

If pairing is a problem, then the server side of the application should be run on the Linux BlueZ stack, which can create an 'agent' process to accept pairing requests. The clients can keep on using the Microsoft (or Widcomm) stack but have to utilize non-standard extensions to JSR-82 in BlueCove to send pairing requests. I'll talk more about this in section 8.

1. Overview of the Client/Server Application

This chapter's example shows how a Bluetooth server 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 connect to the server at once, with the server using a dedicated thread for each one.

The server is a command line application which spends most of its time waiting for client connections. Once a connection has been established, the application task is quite simple: a message sent by the client is echoed back by the server thread, changed to uppercase. Figure 1 shows the server starting up on the "Andrew" device, creating a thread to handle a connection from the "oak" client, and processing two messages.

```

> run EchoServer
Executing EchoServer with BlueCove...
BlueCove version 2.1.0 on widcomm
Device name: ANDREW
Bluetooth Address: 001F81000250
Discoverability set: true
Start advertising echoserver...
Waiting for incoming connection...
Connection requested...
  Handler spawned for client: OAK-F27751CB504
Waiting for incoming connection...
OAK-F27751CB504 --> "hello"
OAK-F27751CB504 <--- HELLO
OAK-F27751CB504 --> "today is Thursday"
OAK-F27751CB504 <--- TODAY IS THURSDAY
    
```

Figure 1. The BlueCove Echo Server.

The client first sends the message "hello", and receives back "HELLO". Then "today is Thursday" arrives at the server, and is also changed to uppercase. Figure 2 shows the client shortly after receiving the second response.

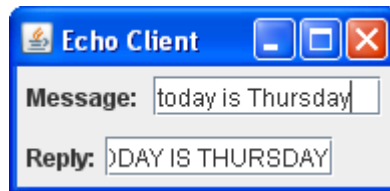


Figure 2. The BlueCove Echo Client.

Figure 3 shows the class diagrams for the server-side of the application. The top-level server is created by the EchoServer class, and each ThreadedEchoHandler thread deals with a separate client connection.

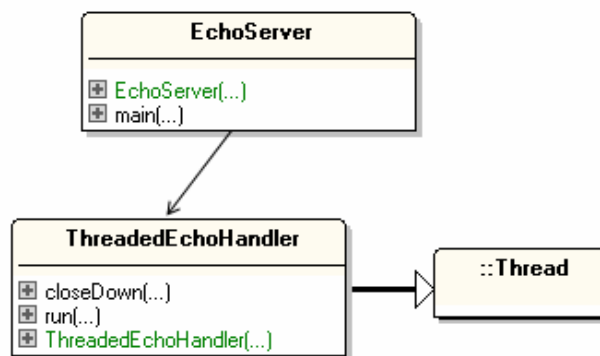


Figure 3. Class Diagrams for the Server.

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

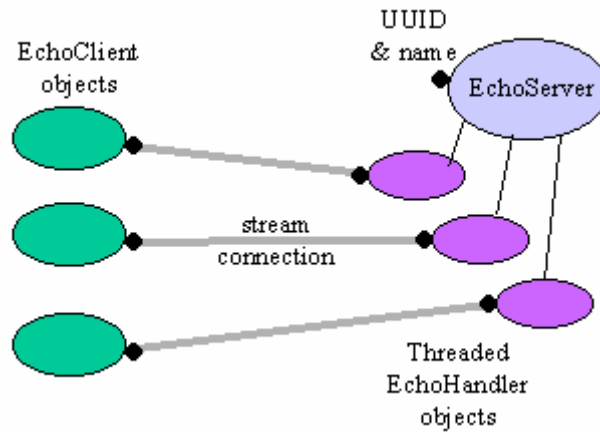


Figure 4. Client and Server Communication.

The stream connection between a client and handler employs 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.

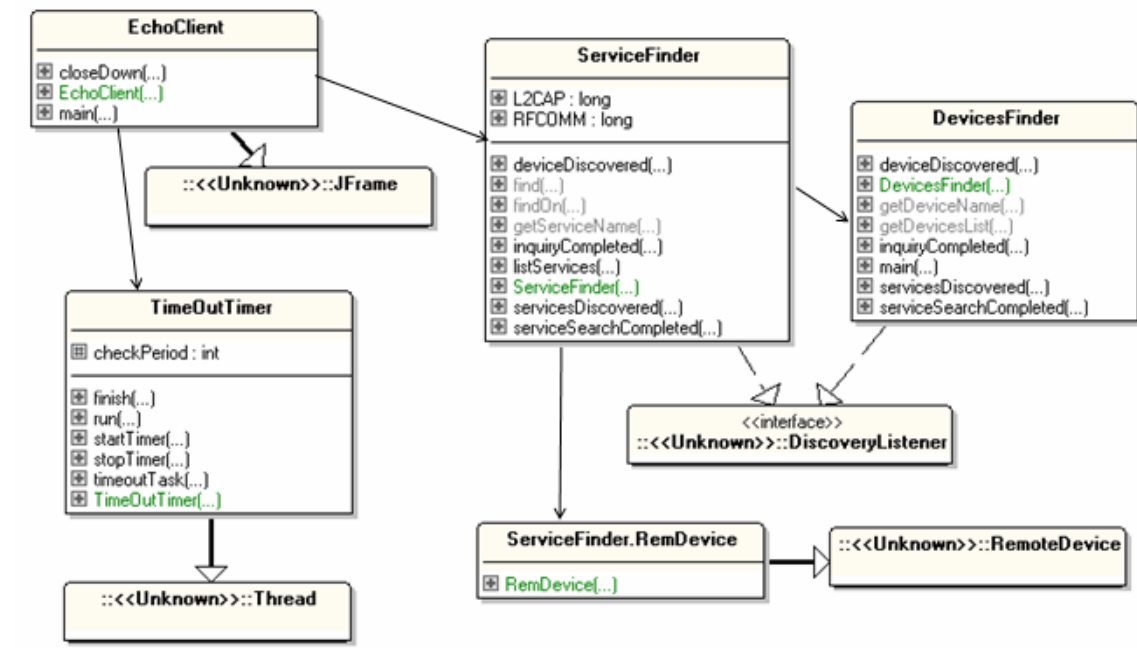


Figure 5. Class Diagrams for the Client.

EchoClient implements the client, which is quite simple mainly because the hard work of device and service discovery are handled by the DevicesFinder and ServiceFinder classes. TimeOutTimer is employed to stop the client waiting forever if a message is not answered. The RemDevice class is utilized by ServiceFinder to create a RemoteDevice object, as explained later.

2. The Server

EchoServer starts by make the server's device discoverable, printing out its name and Bluetooth address. This address can be employed by the client to more directly (and quickly) connect to the server, as I'll discuss when I talk about the findOn() search method in section 6.4.

```
private void initDevice()
{
    try { // make the server's device discoverable
        LocalDevice local = LocalDevice.getLocalDevice();
        System.out.println("Device name: " + local.getFriendlyName());
        System.out.println("Bluetooth Address: " +
            local.getBluetoothAddress());
        boolean res = local.setDiscoverable(DiscoveryAgent.GIAC);
        System.out.println("Discoverability set: " + res);
    }
    catch (BluetoothStateException e) {
        System.out.println(e);
        System.exit(1);
    }
} // end of initDevice()
```

A quick look back at Figure 1 reveals that this “Andrew” device has the Bluetooth address 001F81000250.

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

```
LocalDevice local = LocalDevice.getLocalDevice();
boolean res = 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.

createRFCOMMConnection() is called to create a RFCOMM connection notifier for the server, with the given UUID and name. Both parameters are also used by the client to find the service at search time.

```
// 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 StreamConnectionNotifier server;

private void createRFCOMMConnection()
{
    try {
        System.out.println("Start advertising " + SERVICE_NAME + "...");
        server = (StreamConnectionNotifier) Connector.open(
            "btspp://localhost:" + UUID_STRING +
            ";name=" + SERVICE_NAME + ";authenticate=false");
    }
}
```

```

    catch (IOException e) {
        System.out.println(e);
        System.exit(1);
    }
} // end of createRFCOMMConnection()

```

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

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

I use `localhost` as the hostname. 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". It's a good idea to set the value of the "name" string in lowercase or some stacks won't be able to recognize the service name at discovery time.

The inclusion of "authenticate=false" in the Bluetooth URL may allow the server to bypass explicit pairing with the client, but this depends on the types of devices and stacks being used. In my tests, it did not disable pairing for clients accessing the server from a laptop or netbook.

2.1. Waiting for a Client

The server enters a while-loop inside `processClients()` which spawns a `ThreadedEchoHandler` thread when a client connects to the server

```

// globals
private ArrayList<ThreadedEchoHandler> handlers;
private volatile boolean isRunning = false;

private void processClients()
{
    isRunning = true;
    try {
        while (isRunning) {
            System.out.println("Waiting for incoming connection...");
            StreamConnection conn = server.acceptAndOpen();
            // wait for a client connection
            System.out.println("Connection requested...");

            ThreadedEchoHandler hand = new ThreadedEchoHandler(conn);
            // create client handler
            handlers.add(hand);
            hand.start();
        }
    }
    catch (IOException e) {
        System.out.println(e);
    }
} // end of processClients()

```

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 `StreamConnection` object, which is passed to a `ThreadedEchoHandler` instance so it can deal with the client communication.

2.2. Closing Down

Since the server doesn't have a GUI with a close-box, the server constructor utilizes a shut-down hook to call `closeDown()` when the server is terminated.

```
// in EchoServer's constructor:
Runtime.getRuntime().addShutdownHook(new Thread() {
    public void run()
    {   closeDown(); }
});
```

The server and handler threads are terminated by `closeDown()`.

```
private void closeDown()
{
    System.out.println("Closing down server");
    if (isRunning) {
        isRunning = false;
        try {
            server.close();
        }
        catch (IOException e)
        {   System.out.println(e);   }

        // close down all the handlers
        for (ThreadedEchoHandler hand : handlers)
            hand.closeDown();
        handlers.clear();
    }
} // end of closeDown();
```

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

The handlers are each terminated by calling their `closeDown()` method.

3. The Threaded Echo Handler

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

```
// globals
private StreamConnection conn; // client connection
```

```

private String clientName;

public ThreadedEchoHandler(StreamConnection conn)
{
    this.conn = conn;
    // store the name of the connected client
    clientName = reportDeviceName(conn);
    System.out.println(" Handler spawned for client: " + clientName);
} // end of ThreadedEchoHandler()

private String reportDeviceName(StreamConnection conn)
/* Return the 'friendly' name of the device being examined,
   or "device ??" */
{
    String devName;
    try {
        RemoteDevice rd = RemoteDevice.getRemoteDevice(conn);
        devName = rd.getFriendlyName(false);
    }
    catch (IOException e)
    { devName = "device ??"; }
    return devName;
} // end of reportDeviceName()

```

The information about the 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 devices, because the requested connection may take the total number of connections past the maximum allowed.

3.1. Connecting to the Client

`ThreadedEchoHandler`'s `run()` method creates input and output streams, and starts processing the client's messages. When message processing is finished, the streams and connection are closed down.

```

// globals
private StreamConnection conn;
private InputStream in;
private OutputStream out;

public void run()
{

```

```

try {
    // Get I/O streams from the stream connection
    in = conn.openInputStream();
    out = conn.openOutputStream();

    processMsgs();

    System.out.println(" Closing " + clientName + " connection");
    if (conn != null) {
        in.close();
        out.close();
        conn.close();
    }
}
catch (IOException e)
{ System.out.println(e); }
} // end of run()

```

An `InputStream` and `OutputStream` are extracted from the `StreamConnection` instance, and used in `processMsgs()`.

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).

3.2. Processing Client Messages

The `processMsgs()` method repeatedly 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, so the processing loop exits.

```

// globals
private volatile boolean isRunning = false;

private void processMsgs()
{
    isRunning = true;
    String line;
    while (isRunning) {
        if((line = readData()) == null)
            isRunning = false;
        else { // there was some input
            System.out.println(" " + clientName + " --> \" " +
                               line + "\"");

            if (line.trim().equals("bye$$"))
                isRunning = false;
            else {
                String upper = line.trim().toUpperCase();
                if (isRunning) {
                    System.out.println(" " + clientName + " <--- " + upper);
                    sendMessage(upper);
                }
            }
        }
    }
}

```



```

    }
} // end of processMsgs()

```

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()`.

3.3. 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(clientName + ": 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(clientName + ": Message Read Error");
                return null;
            }
            len += ch;
        }
    }
    catch (IOException e)
    {
        System.out.println("readData(): " + e);
        return null;
    }

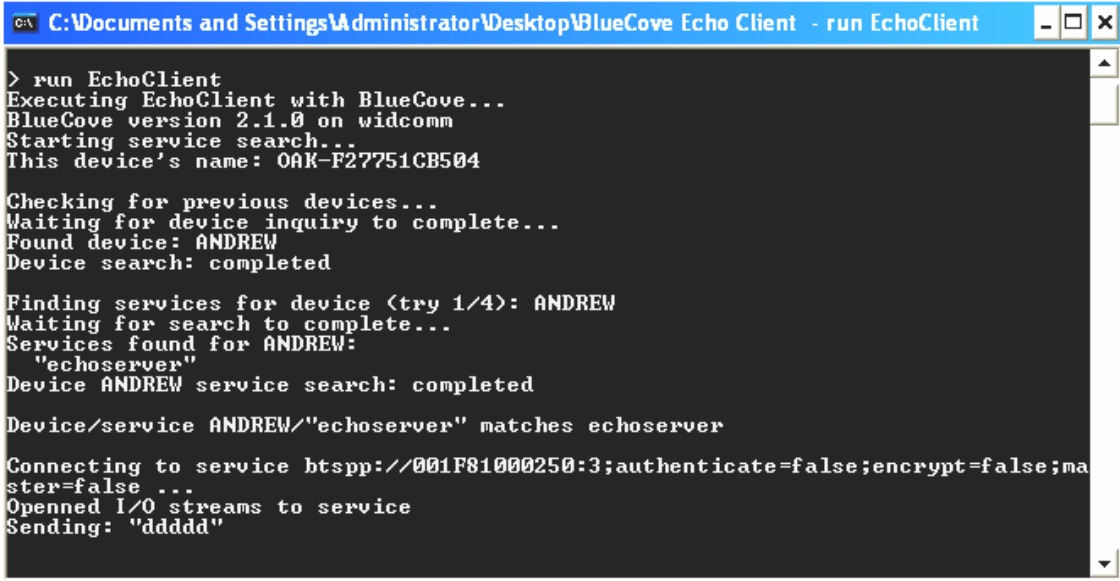
    return new String(data).trim(); // convert byte[] to String
} // end of readData()

```

`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.

3.4. 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:



```
> run EchoClient
Executing EchoClient with BlueCove...
BlueCove version 2.1.0 on widcomm
Starting service search...
This device's name: OAK-F27751CB504

Checking for previous devices...
Waiting for device inquiry to complete...
Found device: ANDREW
Device search: completed

Finding services for device (try 1/4): ANDREW
Waiting for search to complete...
Services found for ANDREW:
"echoserver"
Device ANDREW service search: completed

Device/service ANDREW/"echoserver" matches echoserver

Connecting to service btspp://001F81000250:3;authenticate=false;encrypt=false;ma
ster=false ...
Opened I/O streams to service
Sending: "ddddd"
```

Figure 6. Search Output at Client Start.

The ServiceFinder constructor can be called without a UUID string for the server:

```
ServiceFinder srvFinder =
    new ServiceFinder(ServiceFinder.RFCOMM, SERVICE_NAME);
```

This will typically increase the search time since more matching services will be passed back to the client for processing.

The ServiceFinder class encapsulates the most complex aspects of writing a Bluetooth application – having a client discover devices, and then search each of those devices for relevant services. The next section explains device discovery, while section 6 considers services search.

5. Device Discovery

Figure 7 shows the key stages in the discovery of devices:

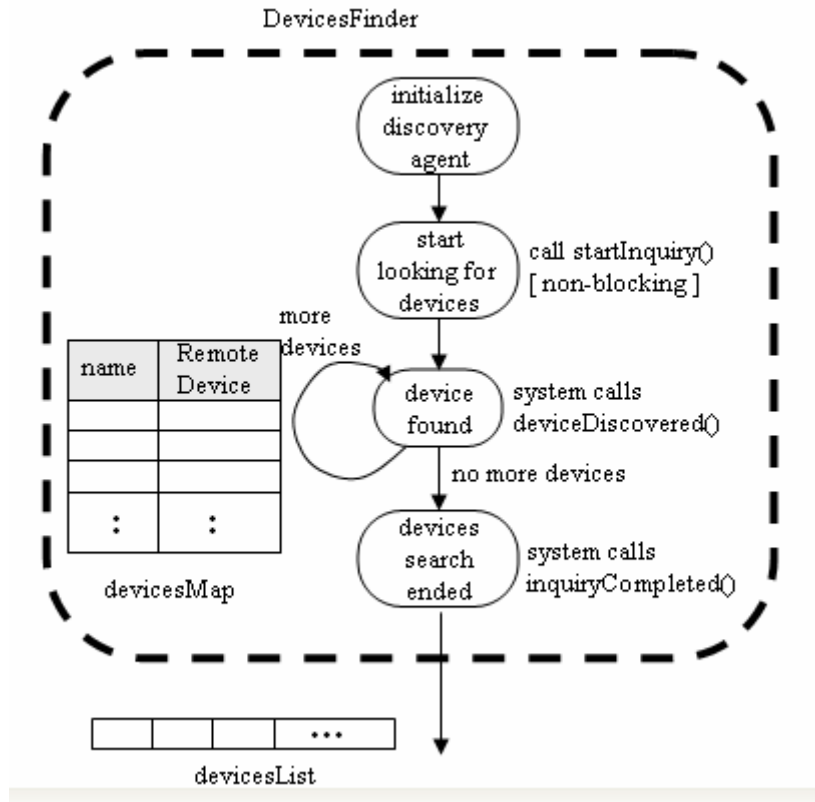


Figure 7. Devices Discovery State Diagram.

The state diagram includes three methods that are utilized in devices search: `DiscoveryAgent.startInquiry()`, `DiscoveryListener.deviceDiscovered()`, and `DiscoveryListener.inquiryCompleted()`. The latter two are implemented by the application, as explained below.

Figure 7 finishes by outputting a list of devices (actually an array of `RemoteDevice` objects), which is passed to the services search described in section 6.

The `DevicesFinder` constructor creates a discovery agent, then starts the devices search by calling `DiscoveryAgent.startInquiry()`:

```

// globals
private HashMap<String,RemoteDevice> devicesMap = null;
    // stores the remote devices found during the device search

private final Object inquiryCompletedEvent = new Object();
    // used to block execution until startInquiry() finishes

public DevicesFinder()
{
    devicesMap = new HashMap<String,RemoteDevice>();
    System.out.println("Starting service search...");
    try {
        // get the discovery agent, by asking the local device
        LocalDevice local = LocalDevice.getLocalDevice();
        System.out.println("This device's name: " +
            local.getFriendlyName());
        agent = local.getDiscoveryAgent();
    }
}
  
```

```

System.out.println("\Checking for previous devices...");
initDevices(DiscoveryAgent.CACHED);
initDevices(DiscoveryAgent.PREKNOWN);

boolean hasStarted =
    agent.startInquiry(DiscoveryAgent.GIAC, this); //non-blocking
if (!hasStarted)
    System.out.println("Device inquiry failed to start");
else {
    System.out.println("Waiting for inquiry to complete...");
    synchronized(inquiryCompletedEvent){
        inquiryCompletedEvent.wait();
    }
}
}
}
catch (Exception e) {
    System.out.println(e);
}
} // end of DevicesFinder()

```

devicesMap is used during the search to hold device name and RemoteDevice information pairs.

The tricky aspect of device discovery is that DiscoveryAgent.startInquiry() is non-blocking, which means that my code must make itself wait until the search has finished. Also, the second argument of startInquiry() is a reference to the DiscoveryListener (this object) whose deviceDiscovered() and inquiryCompleted() methods must be implemented. The search communicates its progress by repeatedly calling deviceDiscovered() as new devices are found, and signals that it has finished by executing inquiryCompleted() (as shown in Figure 7).

After DiscoveryAgent.startInquiry() is started, the DevicesFinder() blocks itself by waiting on a ‘wake-up’ signal for the inquiryCompletedEvent object, which is eventually transmitted by my inquiryCompleted() method:

```

public void inquiryCompleted(int inqType)
// device search has finished
{
    showInquiryCode(inqType);
    synchronized(inquiryCompletedEvent){
        inquiryCompletedEvent.notifyAll(); // wake-up signal
    }
} // end of inquiryCompleted()

```

The DevicesFinder constructor also calls initDevices() to load any cached and preknown RemoteDevice information into devicesMap before embarking on a new devices search:

```

initDevices(DiscoveryAgent.CACHED);
initDevices(DiscoveryAgent.PREKNOWN);

```

Cached devices are remote devices discovered in earlier searches, while preknown devices are those known to be frequently contacted. Many stacks don't support the storage of preknown device information.

```

private void initDevices(int devType)

```

```
// load previous devices into the devices map
{
  RemoteDevice[] oldDevices = agent.retrieveDevices(devType);
  if (oldDevices == null)
    return;
  String devName;
  for (RemoteDevice dev : oldDevices) {
    devName = DevicesFinder.getDeviceName(dev);
    if (devicesMap.get(devName) == null) {
      System.out.println("Found old device info for: " + devName);
      devicesMap.put(devName, dev);
    }
  }
} // end of initDevices()
```

Using cached information may avoid the need for a devices search, but unfortunately it's impossible to quickly determine which cached devices are relevant to the present search. Also, there's no indication of whether a cache is up-to-date.

5.1. Finding a Device

The search initiated by the `DiscoveryAgent.startInquiry()` calls `deviceDiscovered()` each time a device is found. My implementation examines the device's details, and only adds PCs or phones to `devicesMap`.

```
public void deviceDiscovered(RemoteDevice dev, DeviceClass cod)
/* A matching device was found during the device search.
   Only store it if it's a PC or phone. */
{
  String devName = DevicesFinder.getDeviceName(dev);
  System.out.println("Found device: " + devName);

  int majorDC = cod.getMajorDeviceClass();
  int minorDC = cod.getMinorDeviceClass(); // not used in the code

  // restrict matching device to PC or Phone
  if ((majorDC != 0x0100) && (majorDC != 0x0200)) {
    System.out.println("Device not PC or phone, so rejected");
    return;
  }

  if (devicesMap.get(devName) != null)
    System.out.println(" - updating existing info");
  devicesMap.put(devName, dev);
} // end of deviceDiscovered()
```

It's a good idea to do as much device 'filtering' as possible at this stage, to keep `devicesMap` 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. 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, 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");
```

5.2. The End of the Devices Search

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

```
public void inquiryCompleted(int inqType)
// device search has finished
{
    showInquiryCode(inqType);
    synchronized(inquiryCompletedEvent){
        inquiryCompletedEvent.notifyAll(); // wake-up signal
    }
} // 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 subsequent service search only needs a *list* of `RemoteDevices`, not a map, and so `DevicesFinder` includes `getDevicesList()` for converting the `devicesMap` to an array:

```
public RemoteDevice[] getDevicesList()
{ return devicesMap.values().toArray(
    new RemoteDevice[devicesMap.size()] );
}
```

5.3. Making the Devices Search Faster

The devices search may take over 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 `DevicesFinder` in this example.

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

6. Services Search

The `ServiceFinder` constructor stores parameters employed in later searches, and initializes data structures and the search agent.

```
// globals
private Vector<ServiceRecord> servicesVec;
private javax.bluetooth.UUID[] uuids;
private int[] attrSet;
private String serviceStr; // (partial) name of service
private DiscoveryAgent agent;

public ServiceFinder(long protocolUUID, String UUIDStr,
                    String serviceStr)
{
    this.serviceStr = serviceStr;
    uuids = setUUIDs(protocolUUID, UUIDStr);

    attrSet = new int[1];
    attrSet[0] = 0x0100; // service name attribute

    servicesVec = new Vector<ServiceRecord>();
    try {
        // get the discovery agent, by asking the local device
```



```

        LocalDevice local = LocalDevice.getLocalDevice();
        agent = local.getDiscoveryAgent();
    }
    catch (Exception e) {
        System.out.println(e);
    }
} // end of ServiceFinder()

```

The constructor's three input arguments are the Bluetooth protocol (e.g. RFCOMM), the UUID of the service, and its name (or partial name) stored in `serviceStr`. The service UUID is optional, which makes the creation of `ServiceFinder` easier since there's no need to enter a 32 digit hex string, but it also makes the search more time-consuming. More service records will be returned to the client, which all need to be examined.

The search criteria are specified in two arrays: `uuids[]` and `attrSet[]`. `uuids[]` stores the UUIDs for the protocol and service (if it isn't null), and is initialized by `setUUIDs()`:

```

private javax.bluetooth.UUID[] setUUIDs(long protocolUUID,
                                         String UUIDStr)
{
    javax.bluetooth.UUID[] uuids;
    if (UUIDStr != null)
        uuids = new javax.bluetooth.UUID[2];
    else
        uuids = new javax.bluetooth.UUID[1];

    // add the UUIDs for the protocol and service
    uuids[0] = new javax.bluetooth.UUID(protocolUUID);
    if (UUIDStr != null)
        uuids[1] = new javax.bluetooth.UUID(UUIDStr, false);
    return uuids;
} // end of setUUIDs()

```

`attrSet[]` lists any non-standard attributes that should be included in the retrieved service records. In this case, I want the service's name to be added:

```

attrSet = new int[1];
attrSet[0] = 0x0100; // service name attribute

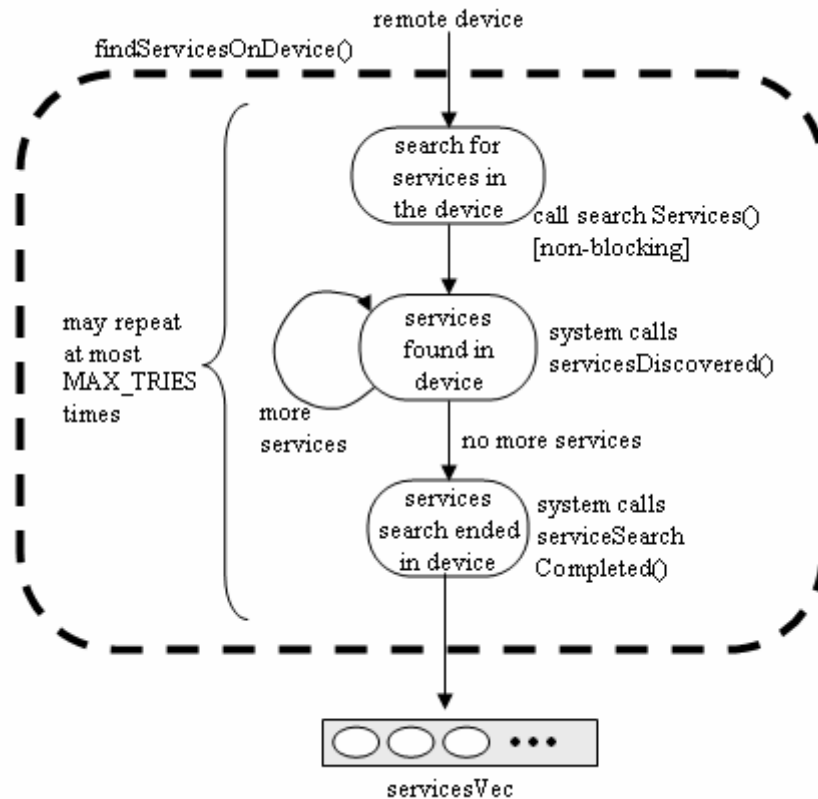
```

Hex values for the many possible UUIDs and attributes are listed at https://www.bluetooth.org/foundry/assignnumb/document/service_discovery.

6.1. The Heart of the Search

As I'll explain shortly, `ServiceFinder` supports two search methods, `find()` and `findOn()`, which utilize private support methods. The most important of these is `findServicesOnDevice()` which collects all the services on a specified remote device matching the service criteria set in the constructor.

The main stages of `findServicesOnDevice()` are shown in Figure 8.

Figure 8. State Diagram for `findServicesOnDevice()`.

The state diagram includes three methods that are utilized in services search: `DiscoveryAgent.searchServices()`, `DiscoveryListener.servicesDiscovered()`, and `DiscoveryListener.serviceSearchCompleted()`. The latter two must be implemented by the application. `searchServices()` initiates a non-blocking services search on a given remote device, which means that my code must make itself wait until the search has finished.

The system communicates its progress by repeatedly calling `servicesDiscovered()` as new service records are found, and signals that the search is over by executing `serviceSearchCompleted()` (as shown in Figure 8).

A major difference in my code from most implementations of service search is a 'retry' mechanism in `findServicesOnDevice()`. In my tests, a service search would often fail to find a service the first time it examined a remote device. For that reason, `findServicesOnDevice()` can repeat the examination at most `MAX_TRIES` times before giving up.

The code for `findServicesOnDevice()`:

```
// globals
private static final int MAX_TRIES = 4;
    // no of times to try calling searchServices()

private Vector<ServiceRecord> servicesVec;
private javax.bluetooth.UUID[] uuids;
private int[] attrSet;
```

```

private DiscoveryAgent agent;

private String devName = null;    // name of device being examined
private boolean searchSucceeded;
private final Object serviceSearchCompletedEvent = new Object();
    // used to block execution until searchServices() finishes

private void findServicesOnDevice(RemoteDevice dev)
/* collect all the services on the remote device, dev, that
   match the service specified in the constructor;
   store matches in the servicesVec
*/
{
    devName = DevicesFinder.getDeviceName(dev);
    servicesVec.clear();    // empty Vector before searching

    searchSucceeded = false;
    int numTries = 0;
    while((numTries < MAX_TRIES) && !searchSucceeded) {
        System.out.println("Finding services for device (try " +
            (numTries+1) + "/" + MAX_TRIES + "): " + devName);
        try {
            int searchID = agent.searchServices(attrSet, uuids, dev, this);
            // non-blocking
            // bluecove restriction: searches are executed sequentially
            if (searchID < 1)
                System.out.println("Service search failed to start");
            else {
                System.out.println("Waiting for search to complete...");
                synchronized(serviceSearchCompletedEvent) {
                    serviceSearchCompletedEvent.wait();    // time to wait
                }
            }
        }
        catch (BluetoothStateException e) {
            System.out.println("Bluetooth state problem: " + e);
            try {
                System.out.println("Wait before retrying...");
                Thread.sleep(1000);    // 1 sec
            }
            catch (InterruptedException ie) {}
        }
        catch (Exception e)    // real error so give up on loop
        { System.out.println(e);
          return;    // due to failure
        }
        numTries++;
    }
} // end of findServicesOnDevice()

```

The call to `DiscoveryAgent.searchServices()` includes the UUIDs and attribute criteria arrays created in the `ServiceFinder()` constructor. The fourth argument is a reference to the `DiscoveryListener` (this object) whose `servicesDiscovered()` and `serviceSearchCompleted()` methods must be implemented here.

After `DiscoveryAgent.searchServices()` is started, `findServicesOnDevice()` blocks itself by waiting on a 'wake-up' signal for the `serviceSearchCompletedEvent` object. This is transmitted by my implementation of `serviceSearchCompleted()`:

```

public void serviceSearchCompleted(int searchID, int respCode)
// Called when the service search has finished
{
    showResponseCode(respCode);
    synchronized (serviceSearchCompletedEvent) {
        serviceSearchCompletedEvent.notifyAll();
        // wake up findServicesOnDevice()
    }
} // end of serviceSearchCompleted()

```

servicesDiscovered() is called whenever a matching service record is found on the device. The information is added to servicesVec.

```

public void servicesDiscovered(int transID,
                               ServiceRecord[] servRecords)
{
    if ((servRecords != null) && (servRecords.length > 0)) {
        System.out.println("Services found for " + devName + ":");
        for (ServiceRecord sr : servRecords) {
            if (sr == null)
                System.out.println("  null service");
            else {
                System.out.println("  \"" +
                                   ServiceFinder.getServiceName(sr) + "\"");
                servicesVec.add(sr);
            }
        }
    }
} // end of servicesDiscovered()

```

6.2. Concurrent Services Searches

Since DiscoveryAgent.searchServices() is non-blocking, it is possible to write a *concurrent* version of findServicesOnDevice() which can search multiple remote devices at the same time. That's the theory at least, but the reality is less exciting.

Most devices and stacks (including BlueCove) don't support concurrent services searches. Attempts to make concurrent calls to searchServices() 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 devices.

Another reason for avoiding concurrent searches is that their coding makes it tricky to implement search termination.

6.3. Using findServicesOnDevice() in find()

A user doesn't carry out searches by directly calling findServicesOnDevice(), instead find() or findOn() are utilized. find() uses DevicesFinder to obtain a list of all

discoverable remote devices, and then searches each in turn for a matching service. The first service record that matches all the search criteria is returned to the user, and no further searching is performed. This strategy is a simple way of speeding up the services search stage, and makes sense for this application since the client only communicates with a single server.

find() is summarized graphically in Figure 9.

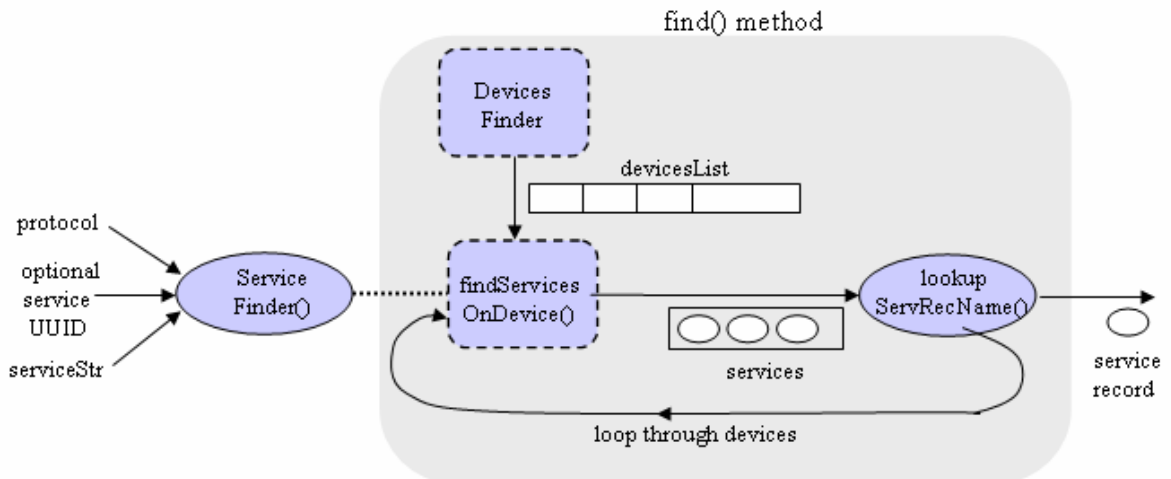


Figure 9. Using the find() Method..

The code for find():

```
public ServiceRecord find()
{
    // find all remote devices
    DevicesFinder devFinder = new DevicesFinder();
    RemoteDevice[] devList = devFinder.getDevicesList();
    System.out.println();
    if ((devList == null) || (devList.length == 0)) {
        System.out.println("No devices found");
        return null;
    }

    // check each device's services in turn
    ServiceRecord sr;
    for (RemoteDevice dev : devList) {
        findServicesOnDevice(dev); // find matching service recs in dev
        if ((sr = lookupServRecName(serviceStr)) != null) {
            // does one of the service rec names match serviceStr?
            return sr;
        }
    }
    return null;
} // end of find()
```

lookupServRecName() utilizes the service name supplied to the ServiceFinder() constructor (serviceStr), to search through the list of service records (servicesVec) retrieved from the current remote device.

```

// global
private Vector<ServiceRecord> servicesVec;

private ServiceRecord lookupServRecName(String servStr)
// return the first service record whose name contains servStr
{
    if ((servicesVec == null) || (servicesVec.size() == 0)) {
        System.out.println("No services found for device "+devName+"\n");
        return null;
    }

    String servName;
    for (ServiceRecord sr : servicesVec) {
        servName = ServiceFinder.getServiceName(sr);
        if ((servName != null) &&
            (servName.contains(servStr))) {
            System.out.println("Device/service " + devName +
                "/\" + servName + "\" matches " + servStr);
            return sr;
        }
    }
    System.out.println("No service found for device " + devName +
        " that matches " + servStr + "\n");
    return null;
} // end of lookupServRecName()

```

6.4. Using findServicesOnDevice() in findOn()

A drawback of find() is the potentially very time-consuming device search at its start. One simple way to speed this up is to supply the address of the required device so device discovery can be bypassed. This is shown in Figure 10.

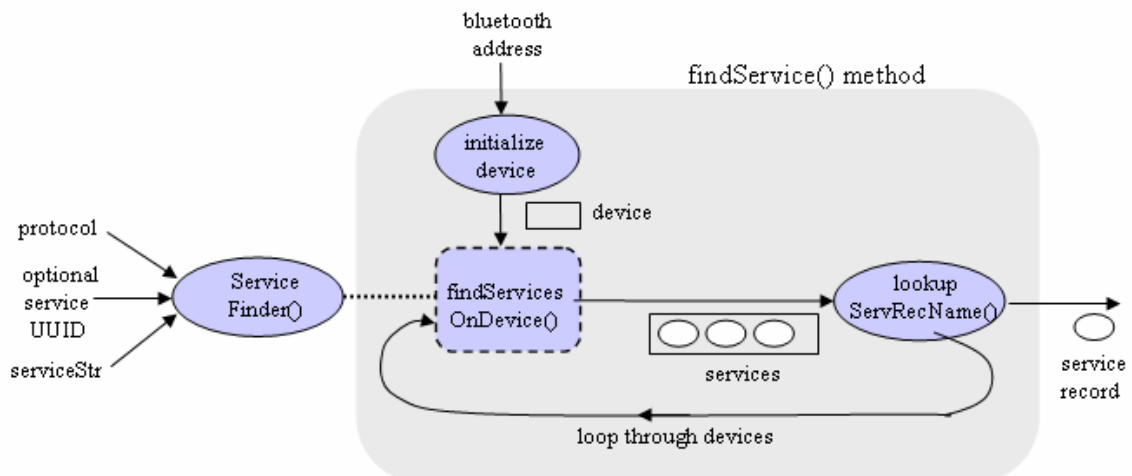


Figure 10. Using the findOn() Method..

The code for findOn():

```
public ServiceRecord findOn(String btAddress)
```

```

{
    // look at remote device at the specified BT address
    RemoteDevice dev = new RemDevice(btAddress);
    System.out.println("Contacting device at BT address " + btAddress);

    findServicesOnDevice(dev); // find matching service recs in dev
    // return service rec whose name matches serviceStr (or null)
    return lookupServRecName(serviceStr);
} // end of findOn()

```

The `RemDevice` class overrides `RemoteDevice` so it can instantiate an object with a Bluetooth address (the constructor in `RemoteDevice` is protected):

```

public class RemDevice extends RemoteDevice
{
    public RemDevice(String btAddress)
    { super(btAddress); }
}

```

A `findOn()` call is supplied with a Bluetooth address, such as:

```
ServiceRecord sr = srvFinder.findOn("001F81000250");
```

This value for the server's Bluetooth address comes from the output when `EchoServer` first starts (see Figure 1).

7. The Client

The GUI part of `EchoClient` consists of a text field for entering a message, and a second text field for the server's response (see Figure 2 for an illustration).

The networking part of the client is started by `makeContact()`:

```

// global
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?

private void makeContact(ServiceRecord servRecord)
{
    // get a URL for the service
    String servURL = servRecord.getConnectionURL(
        ServiceRecord.NOAUTHENTICATE_NOENCRYPT, false);
    if (servURL == null) {
        System.out.println("No service available");
        return;
    }

    System.out.println("\nConnecting to service " + servURL + " .");
    try {
        // connect to the server, and extract IO streams
        conn = (StreamConnection) Connector.open(servURL);

```

```

    out = conn.openOutputStream();
    in = conn.openInputStream();

    System.out.println("Opened I/O streams to service");
    isClosed = false; // i.e. the connection is open
}
catch (Exception ex)
{ System.out.println(ex);
  System.exit(1);
}
} // end of makeContact()

```

makeContact() gets its service record from the find() (or findOn()) service search carried out before the creation of the EchoClient object.

The first argument of 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().

7.1. Sending a Message

A message is sent out using echoMessage(), which waits for an response.

```

// global
private TimeoutTimer timer;
    // used to prevent infinite blocking of message reply waiting

private String echoMessage(String msg)
{
    if (isClosed) {
        System.out.println("No Connection to Server");
        System.exit(1);
    }

    if ((msg == null) || (msg.trim().equals("")))
        return "??"; // empty message
    else {
        if (sendMessage(msg)) { // message sent ok
            timer.startTimer(TIME_OUT);
            String response = readData();
            // wait for response for up to TIME_OUT ms
            timer.stopTimer();

            if (response == null) {
                System.out.println("Server Terminated Link");
                System.exit(1);
            }
            else // there was a response
                return response;
        }
    }
}

```



```

    }
    else { // unable to send message
        System.out.println("Connection Lost");
        System.exit(1);
    }
}
return null;
} // end of echoMessage()

```

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

A serious problem with `echoMessage()` is its waiting for a reply. In my tests, messages were occasionally lost, which prevents `readData()` from returning an answer, thereby blocking `echoMessage()` and the client indefinitely.

I fixed the problem by adding a reply time-out with the help of a `TimeOutTimer` object.

```

TimeOutTimer timer = new TimeOutTimer():
:
timer.startTimer(TIME_OUT);
String response = readData();
                // wait for response for up to TIME_OUT ms
timer.stopTimer();

```

`TimeOutTimer.startTime()` starts a separate thread which causes the client to exit if the specified time limit is exceeded. `TimeOutTimer` is based on code from the article "Handling Network Timeouts in Java" (http://www.javacoffeebreak.com/articles/network_timeouts/).

7.2. 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();

```

8. The Problems of Pairing

I talk about pairing problems in Chapter B3, "Bluetooth Programming Problems" (<http://fivedots.coe.psu.ac.th/~ad/jg/blue3/>), but I'll briefly recap the issues here, and describe some BlueCove-related solutions.

A device's services typically expose it to change by its clients, so those clients must pass some security checks beforehand. These checks begin with pairing, which is triggered when a device receives a connection request from an unknown client. The client sends a PIN number (also called a passkey), which is checked on the server-side. If the number is correct then the service is made available.

Pairing adds some problems to the client/server programming model used in my Echo application. The user must enter a PIN on the client-side, and the server must check it. Pairing is a security feature, so is handled by the Host Controller Interface (HCI). Unfortunately, JSR-82 doesn't have a programming interface to HCI, so there's no way to automate these two tasks from within Java code.

This isn't too bad for the client because the device only displays a dialog box, which most users can deal with (if they can remember the PIN number). The problems are more serious on the server since a dialog box appears there whenever a client wants to connect. This means that a fully automated server application cannot be written in JSR-82 because a person is needed to respond to the dialog boxes.

There are solutions in BlueCove, but they employ non-standard features and JSR-82 extensions. On the client-side, BlueCove offers a `RemoteDeviceHelper.authenticate()` method for sending a PIN number to a server, so the user isn't bothered by a dialog box. This can be done inside a modified version of `makeContact()`:

```
// global
private static final String PASS_KEY = "1234";

private void makeContact(ServiceRecord servRecord)
// modified to carry out pairing
{
    // get a URL for the service
    String servURL = servRecord.getConnectionURL(
        ServiceRecord.NOAUTHENTICATE_NOENCRYPT, false);
    if (servURL == null) {
        System.out.println("No service available");
        return;
    }

    System.out.println("\nConnecting to service " + servURL + " .");
    try {
        // client-side pairing code
        RemoteDevice dev = servRecord.getHostDevice();
        boolean authRes = RemoteDeviceHelper.authenticate(dev, PASS_KEY);
        System.out.println("Pairing response: " + authRes);

        // connect to the server, and extract IO streams
        conn = (StreamConnection) Connector.open(servURL);
        out = conn.openOutputStream();
        in = conn.openInputStream();

        System.out.println("Opened I/O streams to service");
        isClosed = false; // i.e. the connection is open
    }
}
```

```
    }  
    catch (Exception ex)  
    { System.out.println(ex);  
      System.exit(1);  
    }  
} // end of makeContact()
```

On the server-side, it's necessary to use the BlueZ stack on Linux.

In BlueZ version 3, it was possible to set a passkey in the `/etc/bluetooth/hcid.conf` configuration file (e.g. see <http://www.developershome.com/sms/gnokiiExamples.asp>).

This feature was removed in BlueZ version 4, being considered a security risk (see <https://bugs.launchpad.net/ubuntu/+source/bluez-utils/+bug/365779>). Now you need to use "simple-agent" (a Python script), which is in the `/test` directory of the BlueZ source (for more details see the post by gattscharo in <https://bbs.archlinux.org/viewtopic.php?pid=505425>). There's also a C version in `test/agent.c`.

A reasonable summary of ways to do BlueZ server-side pairing can be found in the "Pair" section of the Bluetooth openmoko wiki (http://wiki.openmoko.org/wiki/Manually_using_Bluetooth#Pair).

9. More Information

The BlueCove website (<http://bluecove.org/>) is a great source of information on such matters as which stacks to use on different platforms, installation details, programming examples, a wiki (<http://code.google.com/p/bluecove/wiki/Documentation>), and Java documentation for BlueCove's implementation of JSR-82 (<http://bluecove.org/bluecove/apidocs/>). Another very useful resource is the bluecove-users forum at <http://groups.google.com/group/bluecove-users>

There are many other development kits for Java and Bluetooth – see the list at http://www.javablueooth.com/development_kits.html. General Bluetooth information can be found at <https://www.bluetooth.org/> and <http://www.bluetooth.com>.