Chapter B2. L2CAP Echoing

The echoing client/server application in chapter B1 uses RFCOMM, a stream-based communications API which is implemented on top of the packet-based L2CAP. L2CAP segments the stream data supplied by RFCOMM into packets prior to transmission, and reassembles received packets into larger messages.

This chapter revisits chapter B1's echoing application, replacing the RFCOMM communication layer with L2CAP. The primary reason for doing this is to speed up the communication, by reducing the latency and avoiding the overheads involved in establishing a stream connection.

Another reason for moving to L2CAP is that JSR 82's L2CAPConnection class offers a ready() method which can be used to implement a polling server.

As a consequence, this chapter presents *two* L2CAP versions of the echoing application: the first uses threaded handlers very similar to those employed in chapter B1, while the second utilizes polling.

The changes required to move from RFCOMM to L2CAP, and from a threaded server architecture to one using polling are almost completely invisible to the user. Figure 1 shows the threaded L2CAP server with its client.

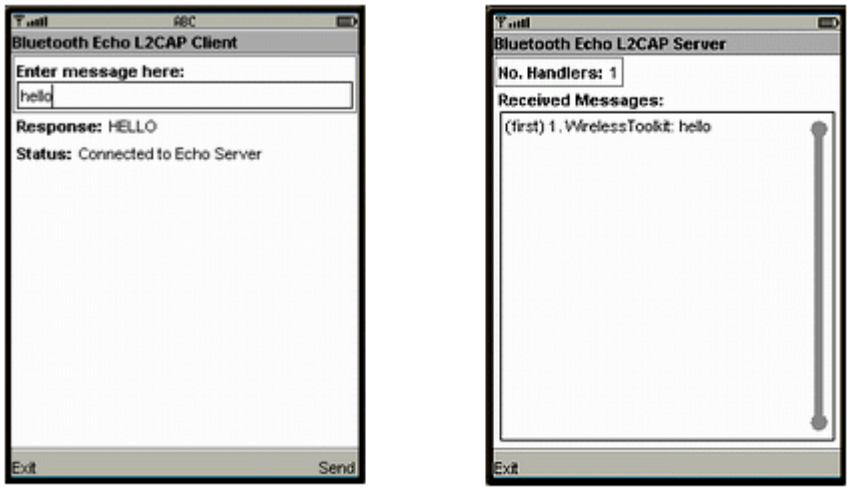

Figure 1. The L2CAP Client and Threaded Server.

Figure 1 is very similar to the figure at the start of chapter B1 for the RFCOMM application. Figure 6 later in this chapter shows the polling L2CAP client and server.

1. An Overview of L2CAP

L2CAP, as defined in the JSR-82 specification, is connection-oriented, which means that it's utilized in an application in much the same way as RFCOMM. (Bluetooth also has a connectionless L2CAP, but JSR-82 doesn't support it.)

A server supplies a suitable formatted URL to Connector.open(), then waits for an incoming client connection by calling acceptAndOpen():

```
private static final String UUID_STRING =
                    "22222222222222222222222222222222";
private static final String SERVICE_NAME = "echolserver";


L2CAPConnectionNotifier server =
             (L2CAPConnectionNotifier) Connector.open(
                        "btl2cap://localhost:" + UUID_STRING +
                        ";name=" + SERVICE_NAME);
L2CAPConnection conn = server.acceptAndOpen();
```

The general format of a L2CAP URL is:

```
btl2cap://<hostname>:<UUID>;<parameters>
```

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".

The call to acceptAndOpen() adds a service record to the device's SDDB, which can be found during a client's services search. The devices and services searches for L2CAP service records only requires a very minor change to the code developed in chapter B1, as I'll explain below.

When a client has a suitable service record, the remote device's URL is extracted from it, and a connection established:

```
String servURL = servRecord.getConnectionURL(
                    ServiceRecord.NOAUTHENTICATE_NOENCRYPT, false);
L2CAPConnection conn = (L2CAPConnection) Connector.open(servURL);
```

Once the connection has been made, the client and server can communicate using the L2CAPConnection send() and receive() methods:

```
public void send(byte[] outMessage);
public int receive(byte[] inMessage);
```

A packet (a message) is encoded as a byte array.

receive() blocks until a packet is available through the L2CAP connection, although the request may time-out, raising an exception.

The L2CAPConnection.ready() method returns true if a packet is waiting to be read from a connection. A server can implement a polling strategy by using ready() to test a collection of the client connections; this greatly reduces the number of threads the server requires, as shown in section 4.

1.1. Negotiating a Maximum Packet Size

The maximum allowed size for messages (packets) is determined when a L2CAP connection is established between two devices with Connector.open().

The negotiation determines two values: ReceiveMTU and TransmitMTU (MTU stands for maximum transmission unit). ReceiveMTU is the maximum number bytes that the local device can receive in a given packet. TransmitMTU is the maximum number of bytes that the local device can send to the remote device in a packet.

The ReceiveMTU for a client and the TransmitMTU for a server are set to the smallest value that both parties can handle. Similarly, the client's TransmitMTU and the server's ReceiveMTU are negotiated to be the smallest packet size they can both process.

For example, if the client can receive a maximum sized packet of 512 bytes, but the server can transmit a packet of size 2048 bytes, then the negotiated client ReceiveMTU and server TransmitMTU will be 512 bytes. Similarly, if the client can send a maximum sized packet of 1024 bytes, but the server can receive a packet of size 2048 bytes, then the negotiated client TransmitMTU and server ReceiveMTU will be 1024 bytes.

This negotiation is illustrated in Figure 2.

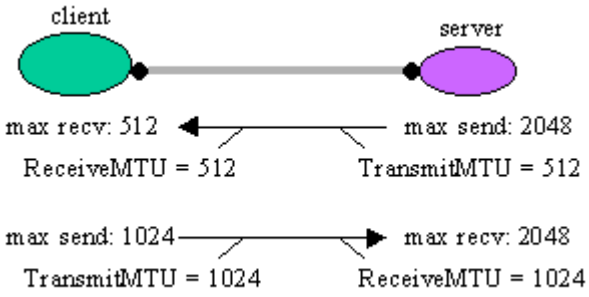

Figure 2. MTU Negotiation.

The client and server can access their negotiated ReceiveMTU and TransmitMTU values by calling getTransmitMTU() and getReceiveMTU():

```
int transmitMTU = conn.getTransmitMTU();
int receiveMTU = conn.getReceiveMTU();
```

These calls should be carried out after the connection has been established.

The packet sizes can be used to set the size of the byte arrays used in the send() and receive() methods.

1.2. Influencing the Negotiation

The ReceiveMTU and TransmitMTU negotiation doesn't require any intervention from the application. However, it might be useful to start the negotiation with

program-specific values when the application wants to negotiate smaller MTU values than the default calculation.

For instance, a smaller negotiated TransmitMTU would require that the transmitted packet be smaller, and smaller packets can be delivered more quickly, thereby reducing latency.

The maximum ReceiveMTU value for the local device is accessible with getProperty():

```
int maxReceiveMTU = Integer.parseInt(
                        LocalDevice.getProperty(
                            "bluetooth.l2cap.receiveMTU.max") );
```

maxReceiveMTU can be utilized in the calculation of user-defined packet sizes which start the MTU negotiation. These numbers are supplied as part of the URL used in Connector.open(). For example:

```
btl2cap://localhost;name=echolserver;ReceiveMTU=512;TransmitMTU=512
```

They become the starting values for ReceiveMTU and TransmitMTU in the negotiation with the remote device.

The initial values should be smaller than (or equal to) the maximum ReceiveMTU. However, there's also a minimum packet size, 48 bytes; Connector.open() will raise an exception if supplied with numbers smaller than this minimum.

1.3. How Fast is L2CAP?

One of the reasons for moving the echo application from RFCOMM to L2CAP is to reduce communications latency. But what's the evidence that L2CAP really does offer better speeds?

The Nokia report, "Games over Bluetooth: Recommendations for Game Developers" (available from the "Documents" section at http://www.forum.nokia.com), states that L2CAP should be used instead of RFCOMM if the data is small, and fast response times are required. However, the numerical data supplied in the report shows no real-world difference in latency times between the two protocols.

The Nokia report includes timing data for clients sending data to a server, where the data was collected by the server, and broadcast back to the clients. When there were 7 clients, the round-trip times varied between 120 and 350ms. When only one client was involved, the times ranged between 20 and 40ms.

Discussions of the relative speeds of RFCOMM and L2CAP at various online forums have included some timing values. L2CAP is faster on more recent phones, with round-trip times between a client and server ranging from 50 to 120ms, while RFCOMM times average over 150ms.

This kind of data should be viewed with some skepticism, since numerous factors may influence the numbers. For instance, round-trip travel times depend on the size of the data being transmitted, since larger packets will require more slots in the Bluetooth frequency band. Packet sizes are also affected by the packet type used by the device. The processing times of the protocol stack, hardware, or firmware, vary considerably from one device to another.

Nevertheless, it does seem that L2CAP is faster than RFCOMM, but the packets must be small.

Unfortunately, I've been unable to run my own timing tests on the examples in this chapter (and chapter B1), since I don't have access to enough Bluetooth devices. I'd be more than happy to receive timing data from kind readers, which I'd add to future versions of this chapter, with my warmest thanks.

2. The Threaded L2CAP Echo Server MIDlet

Figure 3 shows the class diagrams for the threaded L2CAP server, with public and protected methods shown, and the superclasses and interfaces hidden. A glance back to chapter B1 will reveal that it's almost identical to the threaded RFCOMM server, aside from some class name changes.

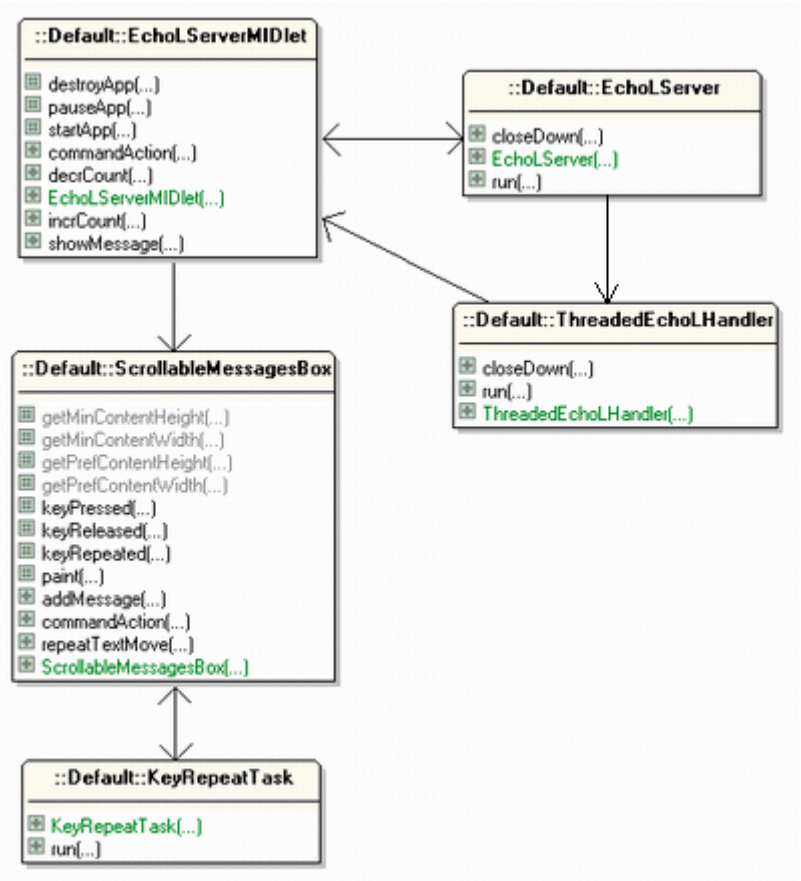

Figure 3. Class Diagrams for the Threaded L2CAP Server.

The similarities aren't really surprising – the GUI elements, ScrollableMessagesBox and KeyRepeatTask, don't need changing, and the threaded model used by EchoLServer and ThreadEchoLHandler is identical to that in EchoServer and ThreadedEchoHandler.

This latter point is highlighted by Figure 4, which shows the communication links between the clients and server handlers:

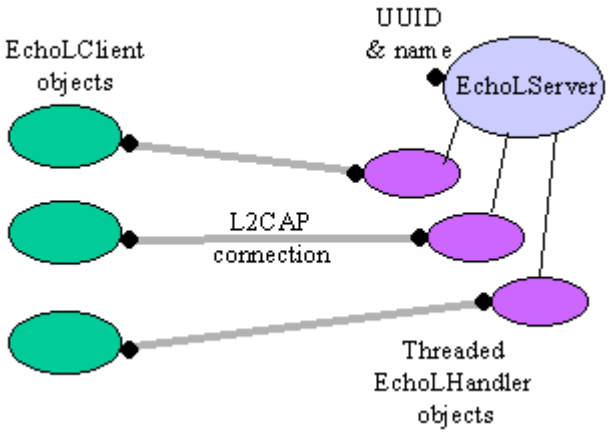

Figure 4. L2CAP Client and Threaded Server Communication.

The only difference between Figure 4, and the same figure in chapter B1 is that the RFCOMM version uses stream connections rather than L2CAP connections.

The important changes caused by the move from RFCOMM to L2CAP are located in EchoLServer and ThreadedEchoLHandler, which I'll describe below.

2.1. The Threaded L2CAP Echo Server

I transformed the RFCOMM EchoServer into the L2CAP EchoLServer by changing the class names for the connection notifier and connection: StreamConnectionNotifier became L2CAPConnectionNotifier, and StreamConnection became L2CAPConnection.

I also modified the UUID and server name:

```
// UUID and name of the echo service
private static final String UUID_STRING =
                    "22222222222222222222222222222222";
private static final String SERVICE_NAME = "echolserver";
```

The constructor creates a L2CAP connection notifier for the server with the UUID and name:

```
// global
private L2CAPConnectionNotifier server;

server = (L2CAPConnectionNotifier) Connector.open(
   "btl2cap://localhost:" + UUID_STRING + ";name=" + SERVICE_NAME);
```

EchoLServer is threaded, so that when it waits for an incoming connection in run(), the GUI isn't forced to stop as well.

```
public void run()
```



```
// Wait for client connections, creating a handler for each one
{
  isRunning = true;
  try {
    while (isRunning) {
      L2CAPConnection conn = server.acceptAndOpen();
      ThreadedEchoLHandler hand =
                    new ThreadedEchoLHandler(conn, ecm);
      handlers.addElement(hand);
      hand.start();
    }
  }
  catch (IOException e)
  {  System.out.println(e);  }
}  // end of run()
```

2.2. The L2CAP Handler

The L2CAP version of the echo handler, ThreadedEchoLHandler, differs from the RFCOMM ThreadedEchoHandler in several ways.

Since the link between the client and handler is an L2CAP connection, packet communication is utilized via send() and receive(). The packet sizes are the negotiated transmit and receive MTU values.

The constructor assigns the MTU numbers to two globals, transmitMTU and receiveMTU, for later use in message passing:

```
// globals
private L2CAPConnection conn; // client connection

// actual sending and receiving packet size
private int transmitMTU, receiveMTU;


public ThreadedEchoLHandler(L2CAPConnection conn,
                                    EchoLServerMIDlet ecm)
{ this.conn = conn;
  this.ecm = ecm;

  try {
    // get negotiated sending and receiving MTUs
    transmitMTU = conn.getTransmitMTU();
    receiveMTU = conn.getReceiveMTU();

    // store the name of the connected client
    RemoteDevice rd = RemoteDevice.getRemoteDevice(conn);
    clientName = getDeviceName(rd);
  }
  catch(Exception e)
  { System.out.println(e); }
} // end of ThreadedEchoLHandler()
```

run() avoids the tricky aspects of packet manipulation by employing readData() and sendMessage().


```
public void run()
/* When a client message comes in, echo it back in uppercase.
   If the client sends "bye$$", or there's a problem, then
   terminate processing, and the handler.
*/
{
  ecm.incrCount();   // tell top-level MIDlet there's a new handler
  isRunning = true;
  String line;
  while (isRunning) {
    if((line = readData()) == null)
      isRunning = false;
    else {  // there was some input
      if (line.trim().equals("bye$$"))
        isRunning = false;
      else {
        ecm.showMessage(clientName + ": " + line);
        String upper = line.trim().toUpperCase();
        if (isRunning)
          sendMessage(upper);
      }
    }
  }
  System.out.println("Handler finished");
  ecm.decrCount();  // remove this handler from the top-level count
}  // end of run()
```

readData() has two main concerns: deciding on the byte array size for the incoming message, and converting the array's contents into a string.

```
private String readData()
// read a message, no bigger than receiveMTU
{
  byte[] data = new byte[receiveMTU];
  try {
    int len = conn.receive(data);
    String msg = new String(data, 0, len);
    return msg;
  }
  catch (Exception e)
  {  System.out.println("readData(): " + e);
     return null;
  }
}   // end of readData()
```

The byte array size is set to be the negotiated ReceiveMTU value.

receive() may time-out after blocking for a while, raising an InterruptedIOException exception. This is caught by an all-purpose catch block, which also handles general IO exceptions.

The main issues with sending a message are converting the string into a byte array, and making sure that a message doesn't exceed the permitted packet size.

```
private boolean sendMessage(String msg)
```



```
// send the message (if it's not too big)
{
  byte[] msgBytes = msg.getBytes();
  if (msgBytes.length > transmitMTU) {
    System.out.println("Message too long, sending \"??\" instead");
    msgBytes = "??".getBytes();
  }
  try {
    conn.send(msgBytes);
    return true;
  }
  catch (Exception e)
  {  System.out.println("sendMessage(): " + e);
     return false;
  }
}  // end of sendMessage()
```

The maximum packet size is the negotiated TransmitMTU, so it's easy to detect if a message is too big. It's harder to know how to deal with the problem, which usually depends on the application's communications protocol. I simply send a "??" message, and report an error.

3. The L2CAP Echo Client MIDlet

Figure 5 shows the class diagrams for the client-side of the L2CAP application.

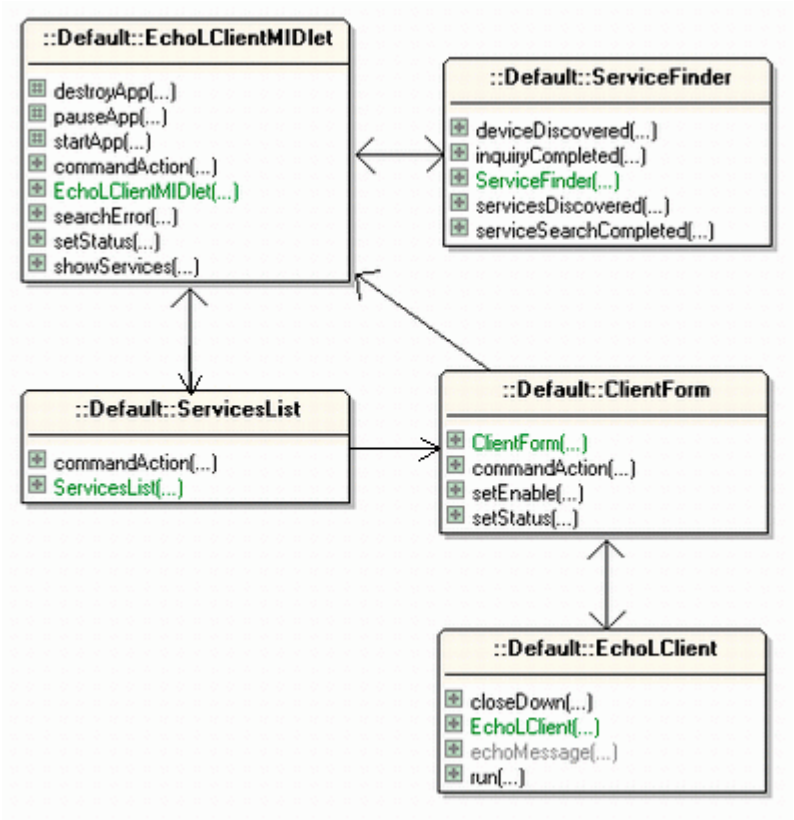

Figure 5. Class Diagrams for the L2CAP Client.

They're almost identical to the class diagrams for the RFCOMM client in chapter B1. The GUI classes, ServicesList and ClientForm are unchanged. EchoLClientMIDlet uses the new UUID and service name, and calls a slightly modified ServiceFinder class.

```
// globals
// UUID and name of the echo service
private static final String UUID_STRING =
                "22222222222222222222222222222222";
private static final String SERVICE_NAME = "echolserver";

// UUID for the L2CAP protocol
private static final long L2CAP_UUID = 0x0100;


// in the constructor
// start searching for services
serviceFinder =
     new ServiceFinder(this, UUID_STRING, SERVICE_NAME, L2CAP_UUID);
```

The ServiceFinder constructor has a new UUID protocol argument, which is assigned to a protocolUUID global variable:

```
private long protocolUUID;      // protocol UUID used by the service
```


protocolUUID is used later in searchForServices() to constrain the services search:

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

  /* Add the protocol UUID to make sure that the matching service
     supports it. */
  uuids[0] = new UUID(protocolUUID);

  // 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()
```

The ServiceFinder class in chapter B1 has the RFCOMM UUID protocol value (0x0003) hardwired into the searchForServices() method:

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

When I revise chapter B1, I'll change its ServiceFinder to be the one used here.

3.1. The L2CAP Echo Client

EchoLClient utilizes L2CAPConnectionNotifier and L2CAPConnection, and packets are manipulated by readData() and sendMessage() (the same methods as in ThreadedEchoLHandler).

The server connection is set up in EchoLClient's run() method since the call to Connector.open() may block for a lengthy period, and shouldn't affect the rest of the MIDlet.


```
// globals
private L2CAPConnection conn;  // for the server
// actual sending and receiving packet size
private int transmitMTU, receiveMTU;


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 = (L2CAPConnection) Connector.open(servURL);

      // get negotiated sending and receiving MTUs
      transmitMTU = conn.getTransmitMTU();
      receiveMTU = conn.getReceiveMTU();

      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 negotiated TransmitMTU and ReceiveMTU values are stored in globals after the connection has been established; they are used in readData() and sendMessage().

When the user enters a message, ClientForm calls EchoLClient's echoMessage() to send the message to the server, and wait for an answer.

```
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()
```

readData() and sendMessage() hide packet manipulation in the same way as in ThreadedEchoLHandler in section 2.2.

4. The Polling L2CAP Application

For a user, the polling L2CAP application is almost identical to the threaded L2CAP and RFCOMM versions. Figure 6 shows that its client and server MIDlets are very similar to the threaded L2CAP application in Figure 1 and the threaded RFCOMM application in Figure 1 of chapter B1.

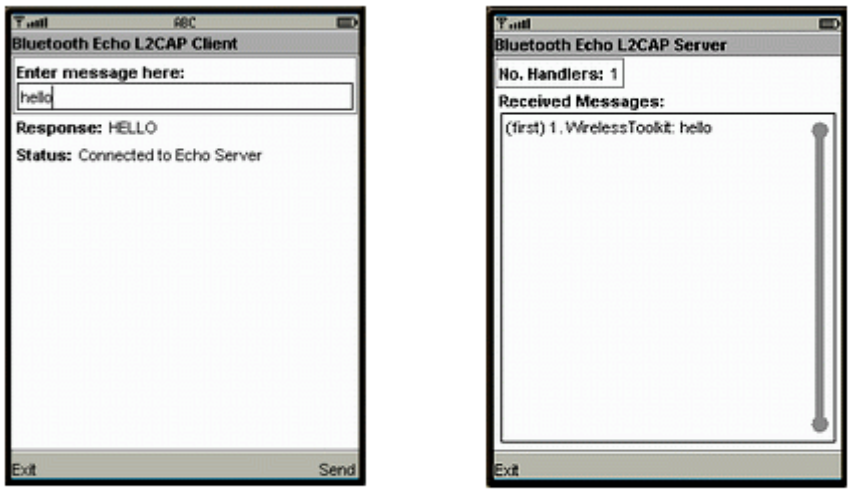

Figure 6. The L2CAP Client and Polling Server.

The client MIDlet, EchoLClientMIDlet, is reused unchanged from the threaded L2CAP application. It can act as a client for either the threaded or polling L2CAP server.

Figure 7 shows how communication is handled between several clients and the polling server.

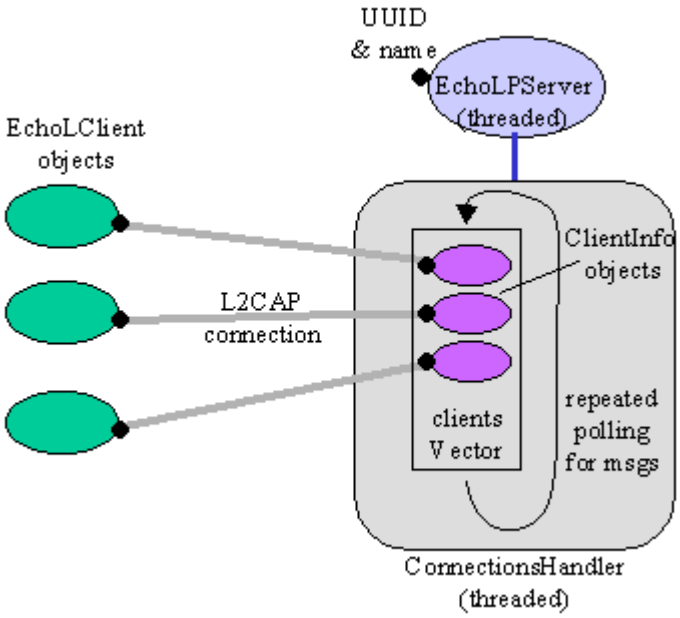

Figure 7. Clients and Polling Server Communication.

Figure 7 should be compared with Figure 4 which shows the communication employed by the threaded server. The threaded server utilizes one handler thread for each client, while the polling server manages with a single ConnectionsHandler.

ConnectionsHandler repeatedly loops through a vector containing client information (each client is represented by a ClientInfo object). It uses L2CAPConnection.ready() to check if a client connection has a message waiting. If ready() returns true, the message is processed, and then the handler moves on to the next ClientInfo object.

The server consists of two threads – the EchoLPServer thread waits for new connections, while connections polling is handled by ConnectionsHandler.

Figure 8 shows the class diagrams for the server MIDlet.

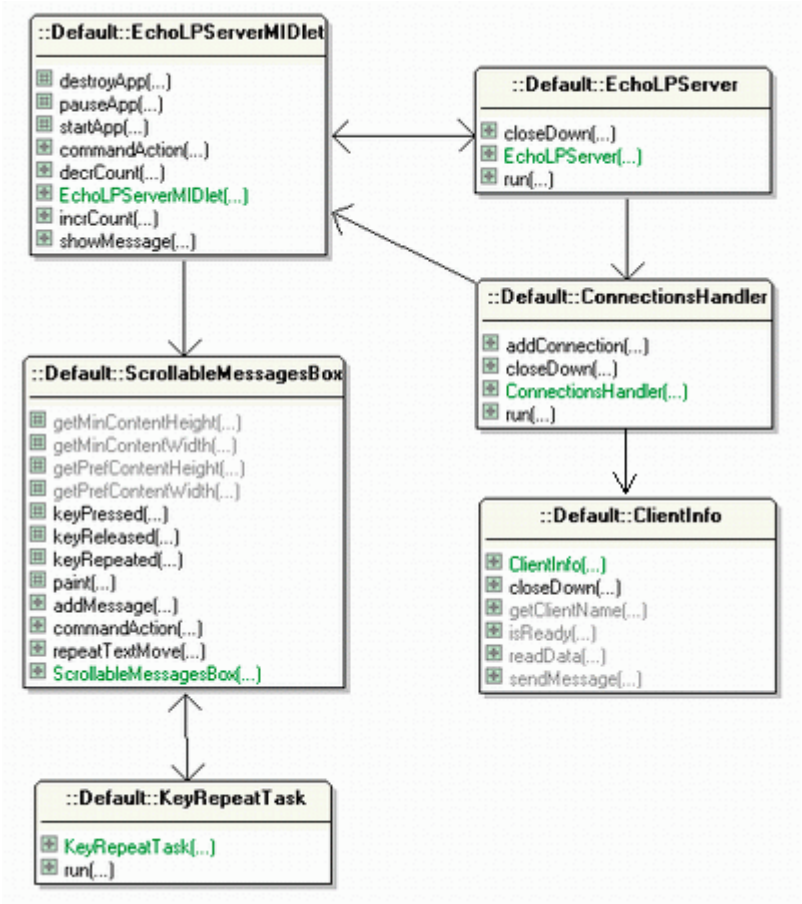

Figure 8. Class Diagrams for the Polling L2CAP Server.

EchoLPServerMIDlet sets up the GUI for the server, using ScrollableMessagesBox and KeyRepeatTask.

4.1. The L2CAP Echo Server Again

EchoLPServer waits for incoming connections, and is the same as EchoLServer (the threaded server), except for the run() method:

```
// global
private ConnectionsHandler handler;

// in EchoLPServer constructor
handler = new ConnectionsHandler(ecm); // ecm is top-level MIDlet
handler.start();


public void run()
// Wait for client connections, and pass it to the handler
{
  isRunning = true;
  try {
    while (isRunning) {
```



```
      L2CAPConnection conn = server.acceptAndOpen();
      handler.addConnection(conn);  // pass connection to handler
    }
  }
  catch (IOException e)
  {  System.out.println(e);  }
}
```

When a new client connection is created, it's passed to the ConnectionsHandler instance, handler, via a call to addConnection(). By comparison, the run() method in EchoLServer creates a new ThreadedEchoLHandler thread for each new connection:

```
// EchoLServer run() code fragment
while (isRunning) {
  L2CAPConnection conn = server.acceptAndOpen();
  ThreadedEchoLHandler hand = new ThreadedEchoLHandler(conn, ecm);
  handlers.addElement(hand);
  hand.start();
}
```

4.2. The Connections Handler

ConnectionsHandler uses a vector to hold client information:

```
private Vector clients;
// vector of ClientInfo objects, one for each client
```

The ClientInfo class stores client connection information, and hides the messy details of sending and receiving messages. A new ClientInfo object is added to the vector when addConnection() is called by EchoLPServer:

```
public void addConnection(L2CAPConnection conn)
{
  ecm.incrCount();   // tell top-level MIDlet there's a new client
  ClientInfo ci = new ClientInfo(conn);  // create client info
  clients.addElement(ci);
}
```

4.2.1. Polling the ClientInfo Objects

run() repeatedly iterates through the ClientInfo objects in the clients vector, examining each one to see if there's a waiting message.

```
// global
private static final int DELAY = 1000;   // ms delay (1 sec)


public void run()
{
  boolean foundMessage = false;
  ClientInfo ci;
  int i = 0;   // index into the clients vector

  while (isRunning) {
    waitWhileEmpty();
```



```
    ci = (ClientInfo) clients.elementAt(i);

    if (ci.isReady()) {   // ci has message waiting to be processed
      foundMessage = true;
      boolean isCliActive = processClient(ci);
      if (!isCliActive) {   // the client connection should be closed
        ci.closeDown();
        clients.removeElementAt(i);   // remove from the vector
        ecm.decrCount();      // remove from the MIDlet's count
      }
      else   // this client is still active
        i++;   // move to next client
    }
    else  // this client didn't have a message ready for processing
      i++;   // move to next client

    if (i == clients.size()) {   // at the end of the vector
      if (!foundMessage) {   // no clients sent msgs during the loop
        try {
          Thread.sleep(DELAY);      // sleep a while
        }
        catch (Exception ex) {}
      }
      i = 0;   // start examining the clients again
      foundMessage = false;
    }
  }
}  // end of run()
```

The call to L2CAPConnection.ready() is done by ClientInfo.isReady(). If it returns true, then the waiting message is processed in processClient(). The method returns a boolean, isCliActive, which is false when the client wishes to break its connection. The connection is closed via a call to ClientInfo.closeDown(), and the object is removed from the clients vector.

Care must be taken with the repeated iteration over the clients vector since the vector's length will vary as clients are added and removed. A counter, i, is used as an index, and is normally incremented after processing a client. However, it's not changed when a ClientInfo object is removed, since the next client will move up to take the position of the departed one. The counter is reset to 0 when it reaches the current end of the vector.

run() keeps cycling through the vector because L2CAPConnection.ready() always returns immediately with true or false. This "busy waiting" strategy causes the ConnectionsHandler thread to consume a lot of resources, but there are two occasions when the thread may sleep.

Whenever a message is found on a connection, the foundMessage boolean is set to true. After looping through the entire vector, foundMessage is tested, and if it's false then it means that none of the connections held a waiting message. In that case, run() sleeps for DELAY milliseconds (1 second).

The other sleeping period occurs when the vector is empty, which is tested by waitWhileEmpty() at the start of each iteration.

```
private void waitWhileEmpty()
// wait until there's some client connections
```



```
{
  while (clients.size() == 0) {  // no clients
    try {
      Thread.sleep(DELAY);      // wait a while
    }
    catch (Exception ex) {}
  }
}
```

The benefit of these "naps" is that run() will utilize the processor less exclusively, giving other threads a chance to execute.

The drawback is that the situations which that trigger sleeping (no messages, no connections) don't supply any information about how long a nap should last. The code utilizes 1 second, but there's no way of knowing if that's too long, too short, or just right. In fact, the suitability of any number will vary during the server's execution.

A much better approach than sleeping is to have the thread suspend until a message arrives on one of its client connections. In J2SE, this behaviour can be implemented using the Selector class, which is part of the NIO API introduced in J2SE 1.4.

Each client channel registers with a selector, and the program waits for a channel message by calling select() with the selector. An example can be found in Chapter 29 of "Killer Game Programming in Java" (http://fivedots.coe.psu.ac.th/~ad/jg/).

select() can also monitor the server socket channel, waking up when a client connects to the server. This functionality means that the server can be implemented with a single thread, as opposed to the two used here.

Hopefully, something like the Selector class will be added to the Bluetooth API in the future.

4.2.2. Processing a Message

run() calls processClient() to handle a message waiting on a connection. processClient () carries out the same task as in the threaded handler: a "bye$$" message signals that the client connection should be closed, while anything else is sent back in uppercase. processClient() uses ClientInfo methods to do the message passing.

```
private boolean processClient(ClientInfo ci)
{
  boolean isCliActive = true;
  String clientName = ci.getClientName();

  String line;
  if((line = ci.readData()) == null)
    isCliActive = false;
  else {  // there was some input
    if (line.trim().equals("bye$$"))
      isCliActive = false;
    else {
      ecm.showMessage(clientName + ": " + line);
      String upper = line.trim().toUpperCase();
      if (isRunning)
        ci.sendMessage(upper);
    }
  }
```



```
  return isCliActive;
}  // end of processClient()
```

readData() and sendMessage() are the same methods that I've employed several times already, except they're now located in the ClientInfo class.

4.2.3. Closing Down the Handler

When the server MIDlet is about to terminate, EchoLPServer calls closeDown() in ConnectionsHandler. closeDown() iterates through its ClientInfo objects, calling their closeDown() methods.

```
public void closeDown()
// close all the clients connections
{
  ClientInfo ci;
  isRunning = false;  // to stop run() loop
  for (int i=0; i < clients.size(); i++) {
    ci = (ClientInfo) clients.elementAt(i);
    ci.closeDown();
  }
  clients.removeAllElements();   // empty the vector
}  // end of closeDown()
```

4.3. Storing Client Information

ClientInfo stores information about a client's connection to the server. Its methods allow messages to be read and sent, and for the connection to be closed.

The constructor employs the L2CAP connection to access the negotiated TransmitMTU and ReceiveMTU values, and the remote device's friendly name.

```
// globals
private L2CAPConnection conn; // client connection
private int transmitMTU, receiveMTU;    // packet sizes
private String clientName;


public ClientInfo(L2CAPConnection c)
{
  conn = c;
  try {
    // get negotiated sending and receiving MTUs
    transmitMTU = conn.getTransmitMTU();
    receiveMTU = conn.getReceiveMTU();

    // store the name of the connected client
    RemoteDevice rd = RemoteDevice.getRemoteDevice(conn);
    clientName = getDeviceName(rd);
  }
  catch(Exception e)
  { System.out.println(e); }
}  // end of ClientInfo()
```


getDeviceName() uses RemoteDevice.getFriendlyName().

ClientInfo's isReady() polls the connection with L2CAPConnection.ready():

```
public boolean isReady()
{ try {
    return conn.ready();
  }
  catch (IOException e)
  {  return false;  }
} // end of isReady()
```

A client connection is closed by closeDown():

```
public void closeDown()
{ try {
    if (conn != null)
      conn.close();
  }
  catch (IOException e)
  {  System.out.println(e);  }
}
```

There's a small chance that closeDown() may be called by the server at the same time that a message is being read or written with readData() or sendMessage(). An exception will be thrown by L2CAPConnection.read() or L2CAPConnection.send(), and caught in the enclosing method.

5. Comparing the Threaded and Polling Approaches

The threaded approach makes the design of the application simpler: each threaded handler manages its own client, leaving the top-level server free to accept new clients. The main drawback occurs when the handlers need to share information about their clients. Due to the handlers' threaded natures, sharing will require synchronized methods so that the data isn't corrupted by concurrent updates. As I'll show in chapter B3, that isn't really that difficult to implement. Another problem may be the number of threads running on the server-side: one for the top-level server, one for each client handler (up to a maximum of seven in a piconet), and one for the MIDlet.

The polling server uses less threads, since all the clients are managed by a single handler. This also makes it easier to share client data, since it's all stored in a single thread. The major drawback of polling is that the connections handler uses resource-intensive busy waiting, and there's no way of having the thread pause, except by sleeping for arbitrary amounts of time. In my opinion, this drawback outweighs the problems with the threaded approach, and so I'll be using the threaded technique in future chapters.

